Bull DPX/20

XTI/XX25
Administrator & User Guide

AIX

ORDER REFERENCE
86 A2 04AP 02

Bull DPX/20

XTI/XX25
Administrator & User Guide

AIX

Software

June 1996

Bull Electronics Angers S.A.
CEDOC
Atelier de Reprographie
331 Avenue Patton
49004 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 04AP 02

Printed in France

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

Trademarks and Acknowledgements

We acknowledge the right of proprietors of trademarks mentioned in this book.

AIX® is a registered trademark of International Business Machines Corporation, and is being used under licence.

UNIX is a registered trademark in the USA and other countries licensed exclusively through X/Open.

About this Book

This document provides detailed information, with a variety of examples, on the Bull-enhanced XTI, Bull implementation of the X/Open Transport Interface (XPG4 XTI) with enhancements.

Warning: When nothing else is specified:

1. Bull-enhanced XTI or XTI refers to the Bull implementation of:
   XTI onto TCP/IP,
   XTI onto OSI Transport,
   XTI onto NetShare,
   XTI onto X.25 (or XX25).
2. Transport Provider refers in fact to:
   a Transport Provider, TCP/IP, OSI Transport or NetShare,
   a Network Provider, concerning XX25.

Who Should Use this Book

This guide is intended for:

- administrators who have to install, configure and maintain XTI,
- programmers who require the services defined by XTI, in order to develop an application.

The programmers who have to port an existing XTI application may refer to *XTI Porting Guide* to have full information about the differences between XPG3 and XPG4.

Before you Begin

Working knowledge of AIX system programming and data communications concepts (especially addressing concepts) is assumed. In particular, working knowledge of:

- TCP/IP is required when XTI is used onto TCP/IP,
- the Reference Model of Open Systems Interconnection (OSI) is required, when XTI is used onto OSI Communication Stack or onto NetShare (RFC 1006),
- X.25 Standard when XX25 is used.

CAUTION:
The reader must be familiar with the XTI concepts developed in *X/Open Transport Interface XPG4 CAE Specification Version 2*

Operating System Level

This document is at Revision 2 level, which applies to AIX Version 4.1

Document Overview

Chapter 1 **Bull-enhanced XTI Overview** presents the product architecture and the Bull enhancements.

Chapter 2 **Installation** lists the prerequisites of Bull-enhanced XTI installation, describes briefly how to install the software and provides a procedure to start with Bull-enhanced XTI.

Chapter 3 **Configurator** describes how to configure the Bull-enhanced XTI parameters: XTI library, XTI Data Base, XTI Option Profiles and XTI Traces.

Chapter 4 **XTI Library Functions**. Each XTI function is described in conformity with *X/Open Transport Interface XPG4 CAE Specification Version 2*. The information specific to Transport Providers (TCP/IP, OSI Transport, NetShare or XX25) has been integrated in the function description and the information specific to Bull-enhanced XTI added under the respective headings:

TCP/IP Implementation Specifics
OSI Implementation Specifics
XX25 Implementation Specifics
Bull Implementation Specifics.

Chapter 5 **XTI Name Server Functions** describes the subroutines of the Name Server library, specific to Bull-enhanced XTI.

Chapter 6 **XTI Name Server Commands** describes the commands specific to Bull-enhanced XTI, especially for use of XTI Name Server, Options and Traces management.

Chapter 7 **Cookbook** provides procedures to prepare a Bull-enhanced XTI application, to manage XTI options and use XTI Traces, and examples of XTI applications with commentaries.

Appendix A **Test Tools** describes the **bench**, **tconnect** and **xtistat** tools provided by Bull-enhanced XTI.

Appendix B **File Formats** describes the XTI Data Base files, used by the XTI Name Server.

Appendix C **Options** list the options available with Bull-enhanced XTI (XTI_GENERIC, ISO_TP, INET_TCP, INET_UDP and INET_IP, X25_NP).

Appendix D **OSI Addressing** describes briefly the OSI network types and associated addresses used as input parameters of the XTI functions.

Appendix E **XX25 Addressing** describes briefly the XX25 addresses types used as input parameters of the XTI/XX25 functions.

Glossary

Index

Revision 02 Modifications

Updates include the support of OSI ConnectionLess Transport Protocol CLTP.

Related Publications

- *X/Open Transport Interface XPG4 CAE Specification Version 2*
  Reference: 40 A2 49AS.
- *XTI Porting Guide*
  Reference: 86 A2 25AP.
- *NetShare User's Guide*
  Reference: 86 A2 95AP.
- *OSI Services Reference Manual*
  Reference: 86 A2 05AQ.
- *HiSpeed WAN Comm. Installation and Service Guide*
  Reference: 86 A1 81WG.
- *XTI Diagnostics Guide*
  Reference: 86 A2 55AJ.
  This document is not delivered with the Bull-enhanced XTI, but may be ordered separately.
- *AIX Installation Guide*
  Reference: 86 A2 60AP.
- *AIX Performance Tuning Guide*
  Reference: 86 A2 72AP.

X/Open Specifications

- X/Open X.25 Programming Interface Using XTI, Preliminary Specifications (XX25)

ISO Standards

- ISO 8072
  Transport Service Definition.
- ISO 8073
  Connection-Oriented Transport Protocol Definition.
- ISO 8208 – The International Standard on information processing systems – Data Communications – X.25 Packet Level Protocol for Data Terminal Equipment (1987).

RFC

- RFC 793
  Transmission Control Protocol.
- RFC 768
  User Datagram Protocol.
- RFC 791
  Internet Protocol.
- RFC 1006
  ISO Transport Services on top of the TCP.

Miscellaneous

AT&T – UNIX SVR4 STREAMS Programmer's Guide.

Table of Contents

Chapter 1. Bull-enhanced XTI Overview

The Bull-enhanced XTI is an Application Programmatic Interface which allows multiple users to communicate using the following communications providers:

- Transport Providers
  - OSI with Connection-Oriented mode of service, usually named as **COTS**,
  - OSI with ConnectionLess mode of service, usually named as **CLTS**,
  - Internet with Connection-Oriented mode of service, **TCP/IP**,
  - Internet with Connectionless mode of service, **UDP/IP**,
  - OSI onto TCP/IP **NetShare (RFC 1006)** (Connection–Oriented mode of service),
- Network Provider
  - X.25 with Connection-Oriented mode of service, usually named as **XX25** (X.25 Programming Interface using XTI).

Warning: Transport and Network Provider are both named as Transport Provider all along this documentation.

The Bull-enhanced XTI provides:

- XPG4 level of functionalities as defined in *X/Open Transport Interface XPG4 CAE Specification Version 2*,
- a **Name Server** to simplify the manipulation of protocol-dependent objects such as addresses and options,
- a **Trace** tool, to help in debug of XTI applications,
- a trouble-shooting tool (**xtistat)** and tests tools (**bench** and **tconnect**), to help in maintenance of XTI applications,
- a fine-grain thread-safe library, which makes easier parallel programming.

The Bull-enhanced XTI is composed of the following components:

- The **XTI Library** is the set of functions defined by X/Open.
  Refer to XTI Library Functions on page 4-1 for more details.
- The **XTI Database** contains Hosts and Services information, Option profiles and Trace levels.
- The **XTI Name Server library** is a set of functions which access to the information contained in the XTI Database.
  Refer to XTI Name Server Functions on page 5-1 for more details.
- The **XTI Configurator**, based on SMIT, allows the management of the XTI Database and the choice of the XTI development environment.
  Refer to XTI Configurator Overview on page 3-1 for more details.
- The **XTI4MOD Streams Module**, in the kernel space is the interface between the XTI library and the transport provider. It is not accessed directly by the XTI applications and is an intrinsic component of the Bull-enhanced XTI implementation,

- The **XPIMOD Streams Module**, in the kernel space is the interface between XTI4MOD and the X.25 provider (XPI_XD). It is not accessed directly by the XTI applications and is an intrinsic component of the Bull-enhanced XTI implementation.

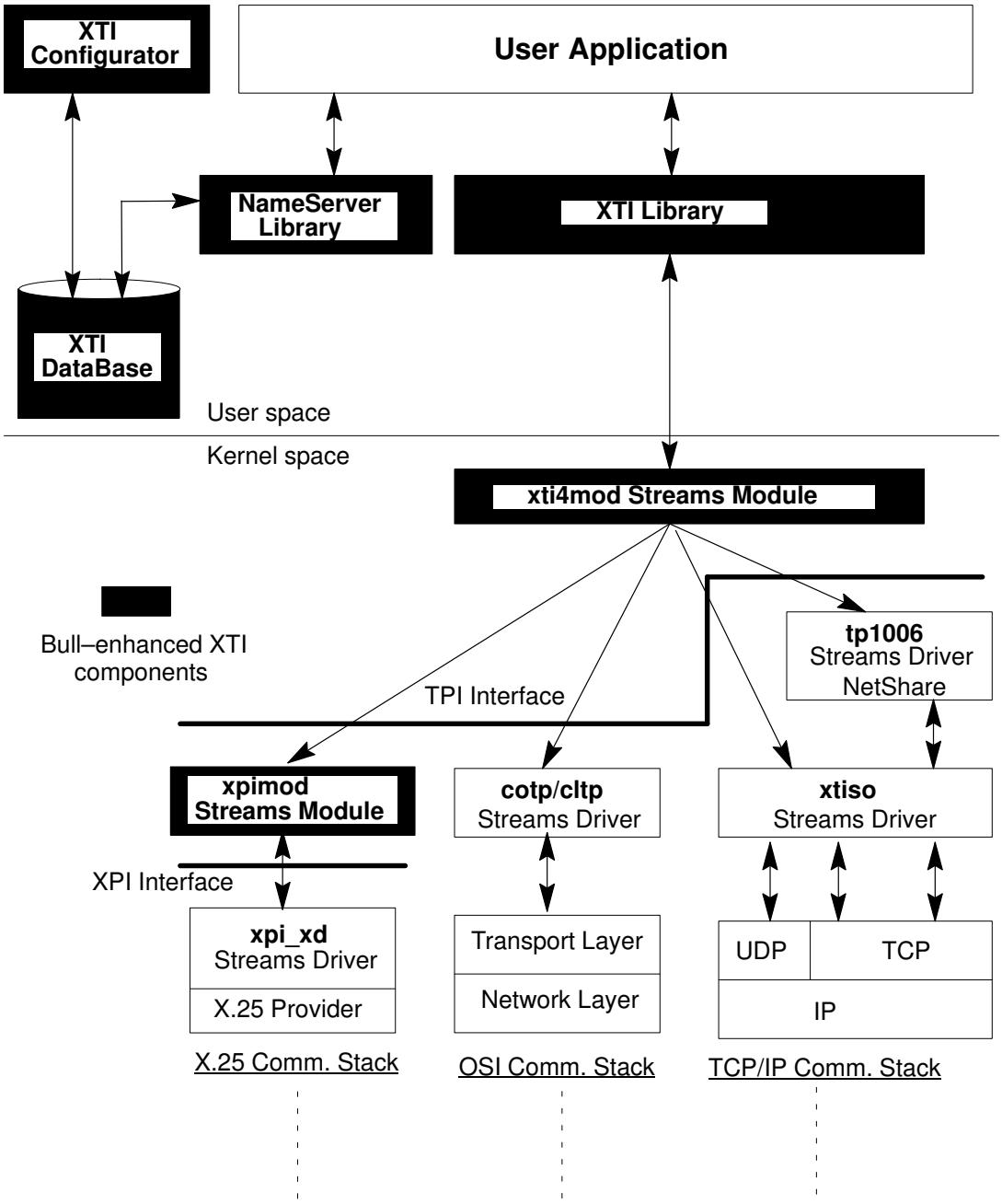


Figure 1. Bull-enhanced XTI Architecture.

Bull-enhanced XTI with Respect to Other Transport Interfaces

Four libraries, offering the same set of subroutines, are available to develop an application accessing Transport Protocol.

1. **Transport Library Interface** (**TLI**), ancestor of **XTI** and used for porting applications developed using the AT&T SystemV-based UNIX operating systems.
   TLI is part of the Base Operating System (BOS) runtime.

2. **X/OPEN Transport Library Interface** (**XTI**), is a library implementation conformant to X/OPEN XPG4 Common Application Environment (CAE) specification. It allows users to communicate using **TCP/IP** and **UDP/IP**.
   XTI is part of the Base Operating System (BOS) runtime.

3. **X/OPEN Transport Library Interface with Bull Enhancements** (named **XTI** too), is a library implementation conformant to X/OPEN XPG4 Common Application Environment (CAE) specification. This **Bull-enhanced XTI** package (**xti_api** LPP) provides in fact two development toolkits:

   – **XTI_ENHANCED**, specified in *X/Open Transport Interface XPG4 CAE Specification Version 2*, allows users to communicate using **TCP/IP** and **UDP/IP**, **OSi** and OSI onto TCP/IP i.e. **NetShare (RFC 1006)**,

   – **XX25**, specified in *X/Open Transport Interface XPG4 CAE Specification Version 2* and in *X/Open Preliminary Specification, X.25 Programming Interface using XTI (August 1994)* allows users to communicate using **X.25**, as well as **TCP/IP** and **UDP/IP**, **OSi** and OSI onto TCP/IP i.e. **NetShare (RFC 1006)**,

   Moreover the Bull-enhanced XTI package provides:

   A Name Server library to simplify the manipulation of protocol-dependent objects such as addresses and options:

   Trace, Trouble-shooting Tool (**xtistat)** and Tests Tools (**bench** and **tconnect**), to help in debug and maintenance of XTI applications.

Notes:

1. **Once the Bull-enhanced XTI is installed on a machine, it is used by default when compiling and linking an XTI application.**
2. The three XTI libraries (basic, XTI_ENHANCED and XX25) may be used on a same machine to compile and link XTI applications. To validate the required library, use the **Bull-enhanced XTI configurator**, on page 3-37, or the **chxti** command, on page 6-12.

Warning:
This document is relative to Bull-enhanced XTI only (XTI_ENHANCED and **XX25)**.

Bull-enhanced XTI Enhancements

The main enhancements brought by Bull-enhanced XTI, respecting conformance with *X/Open Transport Interface XPG4 CAE Specification Version 2* are:

- the XTI Name Server, on page 1-4,
- the XTI Trace, on page 1-5,
- the XTI Tools, on page 1-6.

XTI Name Server

In order to increase portability of XTI applications and to improve their independence with respect to any Transport Provider, the XTI Name Server library contains a set of C primitives, which helps the programmer to not take into account the actual format and representation of transport addresses and options.

The XTI Name Server uses, when existing, Name Server routines and Database of underlying Transport Provider. Its goal is not to replace these name servers but to provide a unified XTI Name Server built on the specific existing name servers (i.e. INET name server on TCP or UDP).

The Transport Providers for which XTI Name Servers allows addresses and options management, are:

- OSI Connection–Oriented,
- OSI ConnectionLess,
- TCP and UDP,
- NetShare (RFC 1006),
- X.25 Connection-Oriented (XX25).

The XTI Name Server manages three types of objects, Services, Hosts and Option Profiles.

XTI Services

A **Service** object defines an association between an Application Name (and aliases) and:

- the Port Number, if TCP/IP,
- the Transport Selector, if OSI or NetShare (RFC 1006),
- the Subsequent Application Identifier (SAI) if XX25,

to be used in order to access to this application from the network.

It must be defined as well by the server which provides it as by the client which uses it.

The Services are saved in:

- */etc/services* file for TCP/IP,
- */etc/xtiservices* file for OSI, NetShare (RFC 1006) and XX25,

and may then be accessed, using the XTI Name Server library, in a transparent way with respect to the Transport Provider.

XTI Hosts

The definition of an XTI **Host** is different according to the Transport Provider used:

- a **TCP/IP Host** object defines an association between a Machine Name (and aliases) and its Internet Address.
- an **OSI Host** object defines a path within the transport to access a remote host. It includes in its definition:
  - the remote Host name,
  - the Network Type,
  - the Remote Address, i.e. the address used to access the remote transport,
  - the Local Address, i.e. the address through which the connection goes out to the remote host.

  According to the network type used to communicate, the addresses may be:
  - SNPA (Sub–Network Point of Attachment),
  - NSAP (Network Service Access Point).
- An **XX25 Host** object defines a path within the network to access a remote host. It includes in its definition:
  - the remote Host name,
  - the Virtual Circuit type,
  - the Remote Address, i.e. the address used to access the remote network layer,
  - the Local Address, i.e. the address through which the connection goes out to the remote host.

Once it is defined in the XTI Data Base, a Host may be accessed, using the XTI Name Server library, in a transparent way with respect to the Transport Provider.

The Hosts are saved in:

- */etc/hosts* file for TCP/IP and NetShare (RFC 1006),
- */etc/xtihosts* file for OSI and XX25.

All the Hosts with which the XTI application has to communicate must be defined.

XTI Option Profiles

An **Option Profile** object is a set of XTI options conformant to the format definition done in *X/Open Transport Interface XPG4 CAE Specification Version 2*. An **Option Profile** is made of a set of items (*level*, *name*, *value*), where:

- *level* identifies the XTI level or a protocol of the transport provider, for instance TCP,
- *name* identifies the option within the level,
- *value* is a value (optional) for the option.

An **Option Profile** defines options, which may be retrieved to build an XTI-XPG4 conformant buffer of options using the *t_getopt* ( ) function.

Refer to **Appendix C. Options** for a complete list of XTI Options available.

The Option Profiles are saved in */etc/xtiopts* file.

XTI Trace

The **XTI Trace** helps the programmer implementing XTI applications for problem determination. It is based on the System Trace facility.

It allows to set trace levels, start and stop trace collections and generate trace reports. Refer to How to Use XTI Traces, on page 7-5, for more details.

XTI Tools

Two types of tools are provided with the Bull-enhanced XTI:

- a Trouble-shooting Tool (**xtistat**) to help in maintenance of XTI applications.
  The **xtistat** command displays global statistics of the XTI activity or the XTI activity of each XTI Transport Endpoint.
- Test Tools (**bench** and **tconnect**) to establish Bull-enhanced XTI performances.
  They may also be used to test if the Bull-enhanced XTI is correctly installed and configured to be used.

Chapter 2. Installation

Here are the sequential tasks to be performed for a correct installation of the Bull-enhanced XTI:

Software Installation

- The software installation must be performed by the system administrator (**root** authority).
- Check in the SRB (Software Release Bulletin) provided with the **xti_api** LPP, that your system conforms to the hardware requirements (disk and memory space).
- The **xti_api** LPP is installed using the standard software installation procedure. Refer to the booklet provided with the Communications Software CD-ROM for more information about installation of the current release.

Package Contents

The **xti_api** LPP is made of the following OPPs:

- **xti_api.com** is mandatory on top of any transport provider. It contains the Bull-enhanced XTI common objects, in particular the XTI and Name Server libraries, the include files, the XTI commands, tools and application examples.
- **xti_api.cotp** is installed in complement of **xti_api.com** to use XTI onto OSI or NetShare (RFC 1006). It contains the OSI objects, in particular the XTI OSI Data Base.

  The OSI Stack LPP (**osi_low**) or (non exclusive) the NetShare (RFC 1006) LPP (**netshare**) is a prerequisite for OPP **xti_api.cotp**.
- **xti_api.xx25** is installed in complement of **xti_api.com** to use XX25.

  The **bullx25** LPP is a prerequisite for OPP **xti_api.xx25**.

Prerequisites

Transport Provider-Independent Prerequisites

Before installing the Bull-enhanced XTI on a machine verify that the following OPP are present:

- `<bos.rte.tty>` to validate use of Streams modules.
- `<bos.adt.base>` if XTI applications have to be developed on this machine.
- `<xlC.C> for` C compiler.
- `<bos.rte.libpthreads>` if thread-safe XTI applications have to be developed and used on this machine.
- `<bos.sysmgt.trace>` if XTI trace facility is used.

Transport Provider-Dependent Prerequisites

If XTI onto TCP/IP

- `<bos.net.tcp.client>` and its prerequisites.

If XTI onto OSI

- `<osi_low.rte>` and OSI stack prerequisites.

If XTI onto NetShare (RFC 1006)

- `<netshare.rte>` and its prerequisites.

If XTI onto X.25 (XX25)

- `<bullx25.xpi>` and its prerequisites.

License

This product uses iFOR/LS encrypted license keys for license management. It supports the 'Nodelocked' license type only. License status is validated only when the product is used, thus permitting installation and configuration without need of the license key.
Refer to the iFOR/LS Installation Notice and Password Order Form delivered with your Communications Product.
Refer to the SRB file for details on how the product uses the license key.

The **xti_api** LPP provides two development toolkits:

- **XTI_ENHANCED**, to use XTI onto TCP/IP, OSI stack and NetShare (RFC 1006),
- **XX25**, to use XTI onto X.25 (XX25) as well as XTI onto TCP/IP, OSI stack and NetShare (RFC 1006).

Each of these development toolkits are licensed independently.

Configuration

To Develop an XTI Application

Define the XTI environments and validate Bull-enhanced XTI by choosing either the XTI development toolkit **XTI–ENHANCED** or the XX25 development toolkit **XX25.**

Warning:
1. The C compiler must be installed to develop an XTI application,
2. If the C compiler was not installed before installing Bull-enhanced XTI, the **/etc/xIC.cfg** file is not updated with the XTI targets (xticc, xticc_r, etc.).
   To integrate them run the command:
   /usr/bin/chxtientry –a –dflt –xpg4 /etc/xIC.cfg

Refer to **XTI Environments Configurator**, on page 3-37.

To Execute an XTI Application

If this application uses the XTI Name Server facilities, the Hosts, Services and Options Profiles must be configured.

An XTI application using the XTI Name Server facilities is developed using objects. For execution, it is easily customized by the end-user defining its own configuration of Transport provider, Hosts, Services and Options.

Configuration of XTI Services

Refer to:

- **How to manage XTI TCP/IP Services**, on page 3-7, if XTI is running onto TCP/IP,
- **How to Manage XTI OSI Services**, on page 3-14, if XTI is running onto OSI or NetShare (RFC 1006),
- **How to Manage XX25 Services**, on page 3-22, if XTI is running onto X.25 (XX25).

A Server application is waiting for incoming calls on a specific transport address:

- a port if XTI onto TCP/IP,
- a TSEL if XTI onto OSI or NetShare (RFC 1006),
- an SAI if XTI onto X.25 (XX25)

Two applications on a same machine cannot wait on the same transport address.

A Client application must know this transport address to be able to access the right application on the remote machine.

1. Choose a transport address (port number, TSEL or SAI, according to the Transport Provider) which is not used, neither on the server machine nor on the client machine.

   The transport addresses already used may be listed using the entry **List all Services** of the XTI configurator.

2. Configure this XTI Service on both the client and server machines using the entry **Add a Service** of the XTI configurator.

Configuration of XTI Hosts

If XTI onto TCP/IP or NetShare (RFC 1006)

Refer to **How to Manage XTI TCP/IP Host** , on page 3-4,

1. Check whether the XTI Host has been previously configured using the command

   ping *host*

2. If the command **ping** is not correct, configure the XTI Host using the entry **Add a Host** of the XTI configurator and declaring:

   the IP address of the Server on the Client machine,

   the IP address of the Client on the Server machine.

If XTI onto OSI

Refer to **How to Manage XTI OSI Hosts**, on page 3-11.

The XTI Hosts have to be defined only for Client applications.

1. Determine which network service (Network type and LSAP) is used to access the server, and consequently which type of addresses NSAP or SNPA have to be used.

   If SNPA, determine them using the command **osiadapterinfo**.

   If NSAP, determine which addresses to use and verify that they have previously configured on OSI stack.

2. Configure this XTI Host using the entry **Add a Host** of the XTI configurator.

If XTI onto X.25 (XX25)

Refer to **How to Manage XX25 Hosts**, on page 3-19.

The XX25 Hosts have to be defined only for Client applications.

1. Determine which Virtual Circuit is used to access the server, and consequently which type of addresses have to be used.
2. Configure this XX25 Host using the entry **Add a Host** of the XTI configurator.

Configuration of Options

Refer to **XTI Option Profile Configurator**, on page 3-25,

1. Consult the specifications of the XTI application in order to know which Options Profiles and Options are used.
2. Determine to which values these options must be set in the specific environment of the running application.
3. Add the necessary Option Profiles using the entry **Add a profile/option** of the XTI configurator.

Chapter 3. Bull-enhanced XTI Configurator

Bull-enhanced XTI Configurator Overview

The XTI configurator allows administrators to:

- define XTI environments, in particular which library is used: AIX–issued XTI, Bull-enhanced XTI or XX25,
- configure the XTI Data Base, i.e. defines Hosts and Services which may be used by the XTI application through the XTI Name Server library.
  This definition is specific to each Transport Provider, TCP/IP (or UDP/IP), OSI, NetShare (RFC 1006), XX25,
- access to the general configuration of the Transport Provider (the XTI configurator is only a bridge to the specific configurator),
- configure the XTI Option Profiles,
- manage the XTI Trace.

Using the XTI Configurator

The XTI configurator is accessed using the **smit** command.

In this configurator description:

- default values, if any, are provided between square braces (”[ ]”),
- mandatory attributes are preceded by an *,
- a sign + at the right-hand end of a line means that the value may be chosen from a list,
- on-line **Help** is available for all dialog fields.

Access Rights

Two levels of access are defined:

- any user can display information concerning the XTI configuration,
- only the XTI Administrator (**root** authority) can modify the XTI configuration (except for **Set User Trace Levels**, user means programmer).

Note: If NLS (National Language Support) is not installed, the messages are displayed in US English language and are not able to be customized.
If NLS is installed, the LANG environment variable must be set; no catalog is selected by default. The following catalog is provided with the XTI product:

US English language catalog (En_US).

XTI Configurator Menu

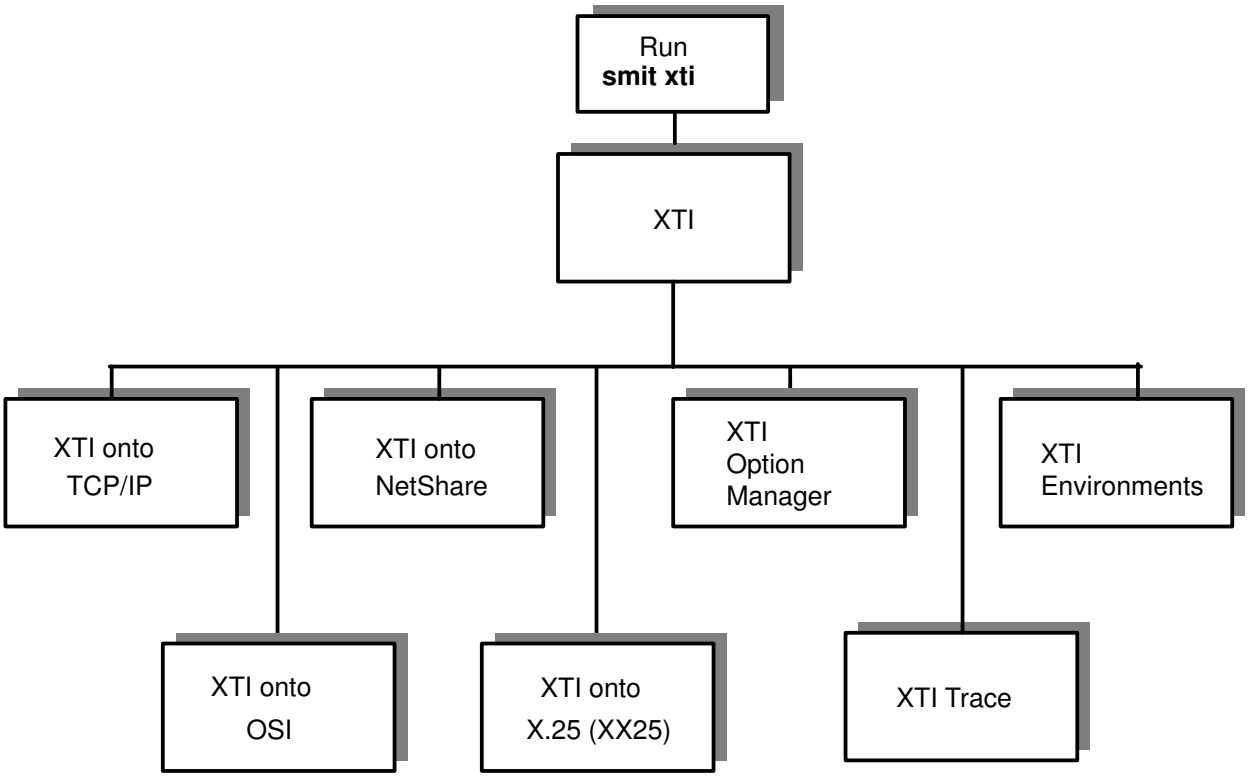


Figure 2. XTI Configurator Menu

Note: The ”XTI onto TCP/IP” menu includes UDP management.

The XTI configurator sub-menus are described in:

- XTI onto TCP/IP, on page 3-3,
- XTI onto OSI, on page 3-10,
- XTI onto NetShare, on page 3-17,
- XTI onto X.25 (XX25), on page 3-18,
- XTI Option Manager, on page 3-25,
- XTI Trace, on page 3-28,
- XTI Environments, on page 3-37.

XTI onto TCP/IP Configurator

Access

Running the command:

smit xtitcp

Description

This menu allows access to the configuration of XTI onto TCP/IP. Select the line corresponding to the type of parameters to be configured.

XTI onto TCP/IP	
XTI/TCP/IP services	see page 3-7
XTI/TCP/IP hosts	see page 3-4
TCP/IP General Configuration	

Note: TCP/IP General Configuration is described in *AIX System Management Guide: Communications and Networks.*

It may be accessed directly using the command: **smit tcpip**.

How to Manage XTI TCP/IP Hosts

Access

Using the XTI onto TCP/IP configurator, run the command:

smit xtitcp

Then select the function:

```
XTI/TCP/IP hosts
```

Description

A **TCP/IP Host** object defines an association between a Machine Name (and aliases) and its Internet Address.

The TCP/IP Hosts are saved in */etc/hosts* file and may be accessed using the XTI Name Server library.

All the machines, with which the local XTI application has to communicate using TCP/IP, must be defined as TCP/IP Hosts.

This menu allows access to the configuration operations of TCP/IP hosts. Select the line corresponding to the action to be performed.

Hosts Table (/etc/hosts)	
List All Hosts	see page 3-5
Add a Host	see page 3-5
Change / Show Characteristics of a Host	see page 3-6
Remove a Host	see page 3-6

Note: This menu may be accessed directly, using the command: **smit hostent**

This menu is a bridge to TCP/IP configuration sub-menus, fully described in *AIX System Management Guide: Communications and Networks.*

Associated Commands

/usr/sbin/hostent

List All Hosts

```
List All Hosts

Address        HostName    HostName     Comment
180.200.20.1    host1
180.200.20.2    host2
180.200.200.10 name.host
```

Note: This menu may be accessed directly, using the command: **smit lshostent**

Lists all the defined TCP/IP Hosts with their attributes, according to the displayed format:

- **Address** is an IP address specified in dotted decimal,
- **HostName** is the name of a Host specified in either relative or absolute domain name format. Multiple Hostnames (or aliases) can be specified,
- a **Comment** may be added.

Add a Host

```
Add a Host

* INTERNET ADDRESS (dotted decimal)                  []
* HOST NAME                                          []
  ALIAS(ES) (if any - separated by blank space)      []
  COMMENT (if any - for the host entry)              []
```

Note: This menu may be accessed directly, using the command: **smit mkhostent**

Allows all attributes, relative to a new Host to be added, to be defined.

Change/Show Characteristics of a Host

```
Change / Show Characteristics of a Host

Host Name or INTERNET Address (dotted decimal)    []            +

Current INTERNET address                          129.183.48.162
New INTERNET ADDRESS (dotted decimal)             []
HOSTNAME                                          [host1]
ALIAS(ES) (if any - separated by blank space)     []
COMMENT (if any - for the host entry)             [#example]
```

Note: This menu may be accessed directly, using the command: **smit chhostent**

Allows the attributes relative to a Host to be displayed or modified.

The name of the host to be displayed or modified is selected from the list and the corresponding attributes to be read or modified are displayed.

Remove a Host

```
Remove a Host

Host Name or INTERNET Address (dotted decimal)     []           +
```

Note: This menu may be accessed directly, using the command: **smit rmhostent**

Removes a Host from the list of defined Hosts.

How to Manage XTI TCP/IP Services

Access

Using the XTI onto TCP/IP configurator, run the command:

smit xtitcp

Then select the function:

```
XTI/TCP/IP services
```

Description

A **TCP/IP Service** object defines an association between a Server-Application Name (and aliases) and the Port Number to be used, in order to access to this application from the network.

It must be defined both by the server which provides it as well as by the client who uses it.

The TCP/IP Services are saved in */etc/services* file and may be accessed using the XTI Name Server library.

The XTI TCP/IP Services, which are part of the XTI product, are automatically defined in the XTI DataBase at the XTI installation:

- *cots_server*, *select_server*, *poll_server*, *benchd* and *tconnectd* for TCP,
- *clts_server* and *benchd* for UDP.

Afterwards, new XTI TCP/IP Services (user applications) can be defined in the XTI DataBase.

This menu allows access to the configuration operations of TCP/IP services. Select the line corresponding to the action to be performed.

Services (/etc/services)	
List All Services	see page 3-8
Add a Service	see page 3-8
Change / Show Characteristics of a Service	see page 3-9
Remove a Service	see page 3-9

Note: This menu may be accessed directly, using the command: **smit inetserv**

This menu is a bridge to TCP/IP configuration sub-menus, fully described in *AIX System Management Guide: Communications and Networks.*

Associated Commands

cat /etc/services | sed –e '/^#/d'
List All Services

/usr/sbin/chservices
Add a Service
Change/Show Characteristics of a Service

List All Services

```
 List All Services

Service         Port/Protocol  Aliases                      Comment
cots_server     20001/tcp      COTS_SERVER     COTS_server     #XTI NS config
select_server   20002/tcp      SELECT_SERVER   SELECT_server   #XTI NS config
poll_server     20003/tcp      POLL_SERVER     POLL_server     #XTI NS config
benchd          20004/tcp      BENCHD          xtibenchd       #XTI NS config
tconnectd       20005/tcp      TCONNECTD       xticonnectd     #XTI NS config

clts_server     20001/udp      CLTS_SERVER     CLTS_server     #XTI NS config
benchd          20004/udp      BENCHD          xtibenchd       #XTI NS config
```

Note: This menu may be accessed directly, using the command: **smit lsservices**

Lists all the defined TCP/IP Services with their attributes, according to the displayed format:

- **Service** is the official Internet Service name,
- **Port** is the Port Number,
- **Protocol** indicates the protocol used (tcp or udp),
- **Aliases** is the list of nicknames used for this Service,
- a **Comment** may be added.

The example lists the services defined at the XTI installation.

Add a Service

```
 Add a Service

* Official Internet SERVICE Name              []
* Transport PROTOCOL                          tcp              +
* Socket PORT number                          []                 #
  Unofficial Internet SERVICE NAMES           []
  (separate names with blanks)
```

Note: This menu may be accessed directly, using the command: **smit mkservices**

Allows all attributes, relative to a new Service to be added, to be defined.

Change/Show Characteristics of a Service

```
Change / Show Characteristics of a Service

* Internet Service Name                          []              +

  Old Internet Service Name                     cots_server
  Old protocol                                  tcp
  Old Socket PORT Number                        20001
  New Internet SERVICE Name                     [cots_server]    +
  New PROTOCOL                                   tcp             +
  Socket PORT number                             [20001]          #
  Unofficial Internet SERVICE NAMES              []
  (separate names with blanks)
```

Note: This menu may be accessed directly, using the command: **smit chservices**

Allows attributes relative to a Service to be displayed or modified.

The name of the service to be displayed or modified is selected from the list and the corresponding attributes to be read or modified are displayed.

Remove a Service

```
 Remove a Service

 * Internet Service Name                          []           +
```

Note: This menu may be accessed directly, using the command: **smit rmservices**

Removes a Service from the list of defined Services.

XTI onto OSI Configurator

Access

Running the command:

smit xtiosi

Description

This menu allows access to the configuration of XTI onto OSI. Select the line corresponding to the type of parameters to be configured.

XTI onto OSI	
XTI/OSI services	see page 3-14
XTI/OSI hosts	see page 3-11
OSI General Configuration	

Note: OSI General Configuration is described in *OSI Services Reference Manual*.

It may be accessed directly using **smit OSIconf**.

How to Manage XTI OSI Hosts

Access

Using the XTI onto OSI configurator, run the command:

smit xtiosi

Then select the function:

```
XTI/OSI hosts
```

Description

An **OSI Host** object defines a path within the transport to access a remote host. The description of an **OSI Host** includes the:

- remote **Host** name,
- **Network Type**,
- **Remote Address**, i.e. the address used to access the remote transport,
- **Local Address**, i.e. the address through which the connection goes out to the remote host.

According to the type of communications network, the addresses may be:

- NSAP (Network Service Access Point),
- SNPA (Sub–Network Point of Attachment).

The OSI Hosts are saved in */etc/xtihosts* file and may be accessed using the XTI Name Server library.

This menu allows access to the configuration operations of XTI/OSI hosts. Select the line corresponding to the action to be performed.

XTI/OSI hosts	
List all hosts	see page 3-12
Add a host	see page 3-13
Change/Show characteristics of a host	see page 3-13
Remove a host	see page 3-13

Note: This menu may be accessed directly, using the command: **smit xtihost**

Associated Commands

xtihost Manages OSI Hosts in the XTI Data Base

Note: A complete description and syntax of the **XTI/OSI Hosts** attributes described below can be found in **xtihost** command on page 6-2.

List All Hosts

```
Host         Remote address  Local  address   Network Type  Lsa  Aliases

localhost    0x02                             CLNS          OSI
h_x25_pvc                    "PVCone"         CONS/WAN/PVC  OSI  new_type example_6
h_x25_svc    138000000000002 138000000000001  CONS/WAN/SVC  OSI  old_nt_1 example_1
h_nullip_osi 0x02608c222222  0x02608c111111   I_CLNS/LAN    OSI  old_nt_4 example_4
h_nullip_dsa 0x003001020304  0x003001020304   I_CLNS/LAN    DSA  old_nt_2 example_2
h_fullip     0x0a0b0c0d                       CLNS          OSI  old_nt_3 example_3
h_spee       0xabef                           SPEE          OSI  old_nt_5 example_5
localhost    0x12                             CLNS          OSI
```

Lists all the defined XTI/OSI Hosts with their attributes, according to the displayed format:

- **Host** is the remote Host name.
- **Remote Address** is the address used to access the remote transport.

  The value meaning depends on the **Network Type**, it may be:

  – a Network Service Access Point (NSAP),

  – a Sub-Network Point of Attachment (SNPA), X.121 or MAC address.
- **Local Address** is the address through which the connection goes out to the remote host.

  The value meaning is the same as for **Remote Address.**
- **Network Type** identifies the network used. It may be:

  CONS/WAN/PVC
  COTS over CONS on Permanent Virtual Circuit.
  The local address is the name of the local PVC to use; the remote address is not significant and should be left blank.

  CONS/WAN/SVC
  COTS over CONS on Switched Virtual Circuit.
  The local and remote addresses are the Sub-Network Point of Attachment (SNPA), in this case the X.121 addresses.

  I_CLNS/LAN
  COTS over Inactive CLNS.
  The remote and local address are the Sub-Network Point of Attachment (SNPA), in this case the MAC addresses.
  There are two modes of use for this network type, depending on **LSAP**:
  if OSI: I_CLNS/LAN Full OSI conformance,
  if DSA: I_CLNS/LAN Non Full OSI conformance.

  CLNS COTS over CLNS on LAN and WAN (Full OSI conformance).
  The remote address is the Network Service Access Point (NSAP); the local address is optional.

  SPEE COTS over CONS on WAN or COTS over CLNS on LAN.
  The remote address is the Network Service Access Point (NSAP), the local address is optional.
- **LSAP**: is the Link Service Access Point used. It is significant only on an **I_CLNS/LAN** network and may be:

  – OSI

  – DSA.
- **Aliases**: are alternative names for *Host* (maximum 2).

Add a Host

```
 Add a Host

* Host name:                                 []
  Local  address (NSAP/SNPA):                []           +
  Remote address (NSAP/SNPA):                []
* Network type:                              CLNS         +
* Link Service Access Point (LSAP)           OSI          +
* Protocols (COTS-CLTS)                      COTS
  Aliases:                                   []
```

Note: This menu may be accessed directly, using the command: **smit xtihostadd**

Protocol is the Protocol Name, as defined in the XTI database */etc/xtiprotocols)*. Can be *tpid_osi_cots* or *tpid_osi_clts*.

Allows all the attributes, relative to a new Host to be added, to be defined.

Change/Show Characteristics of a Host

```
 Change / Show Characteristics of a Host

 Name of the existing host:                                      +

* Old Name:                                   h_nullip_osi
  New host name:                             [h_nullip_osi]
  New local  address (NSAP/SNPA):            [0x02608c111111]    +
  New remote address (NSAP/SNPA):            [0x02608c222222]    +
  New Network type:                           I_CLNS/LAN         +
  New Link Service Access Point (LSAP):       OSI                +
  Aliases:                                   old_nt_4  example_4
```

Note: This menu may be accessed directly, using the command: **smit xtihostchg_sel**

Allows the attributes relative to a Host to be displayed or modified.

The name of the host to be displayed or modified is selected in the list and the corresponding attributes to be read or modified are displayed.

Remove a Host

```
 Remove a Host

Name of the existing host:                                       +
```

Note: This menu may be accessed directly, using the command: **smit xtihostrem_sel**

Removes a Host from the list of defined Hosts.

How to Manage XTI OSI Services

Access

Using the XTI onto OSI configurator, run the command:

smit xtiosi

Then select the function:

```
XTI/OSI services
```

Description

An **XTI/OSI Service** object defines an association between a Server-Application Name (and aliases) and the Transport Selector to be used in order to access to this application from the network.

It must be defined both by the server which provides it as well as by the client who uses it.

The XTI/OSI Services are saved in */etc/xtiservices* file and may be accessed using the XTI Name Server library.

The XTI OSI Services, which are part of the XTI product, are defined in the XTI DataBase automatically at the XTI installation:

cots_server, *clts_server, select_server*, *poll_server*, *benchd* and *tconnectd*.

Afterwards, new XTI OSI Services (user applications) can be defined in the XTI DataBase.

This menu allows access to the configuration operations of OSI services. Select the line corresponding to the action to be performed.

XTI/OSI services	
List All Services	see page 3-15
Add a Service	see page 3-15
Change / Show Characteristics of a Service	see page 3-16
Remove a Service	see page 3-16

Note: This menu may be accessed directly, using the command: **smit xtiserv**

Associated Command

xtiserv Manages XTI/OSI Services in the XTI Data Base

Note: A complete description and syntax of the **XTI/OSI Services** attributes can be found in **xtiserv** command on page 6-5.

List All Services

```
 List All Services
Service        TSEL-SAI/Proto                  Aliases

cons_server   0xC4020301/npid_x25_cons CONS_SERVER          CONS_server
cots_server   0xC4020302/npid_x25_cons COTS_SERVER          COTS_server
select_server 0xC4020303/npid_x25_cons SELECT_SERVER        SELECT_server
poll_server   0xC4020304/npid_x25_cons POLL_SERVER          POLL_server
benchd       0xC4020305/npid_x25_cons BENCHD                xtibenchd
tconnectd     0xC4020306/npid_x25_cons TCONNECTD            xtitconnectd
cots_server   0x01020301/tpid_osi_cots COTS_SERVER          COTS_server
select_server 0x01020302/tpid_osi_cots SELECT_SERVER        SELECT_server
poll_server   0x01020303/tpid_osi_cots POLL_SERVER          POLL_server
benchd       0x01020304/tpid_osi_cots BENCHD                xtibenchd
tconnectd     0x01020305/tpid_osi_cots TCONNECTD            xtitconnectd
clts_server   0x01020306/tpid_osi_clts CLTS_SERVER          CLTS_server
```

Lists all the defined OSI Services with their attributes, according to the displayed format:

- **Service** is the Service name,
- **TSEL** is the OSI Transport SELector,
- **Protocol** is the Protocol name, as defined in the XTI Database (*/etc/xtiprotocols*)

  Can be equal to *tpid_osi_cots, tpid_osi_clts* or *npid_x25_cons.*
- **Aliases** are nicknames for Service name (maximum 2).

The example lists the services defined at the XTI installation.

Add a Service

```
 Add a Service

* Name:                                          []
* Transport SELector (TSEL):                     []
* Protocol:                                      tpid_osi_cots
  Aliases:                                       []
```

Note: This menu may be accessed directly, using the command: **smit xtiservadd**

Allows to define all the attributes relative to a new Service to be added.

The name of the Protocol is selected from the list. *tpid_osi_cots* or *tpid_osi_clts* can be selected.

Change/Show Characteristics of a Service

```
Change / Show Characteristics of a Service

Name of the existing service:                                    +

* Old name:                                    cots_server
* Protocol:                                    tpid_osi_cots
  New name:                                    [cots_server]
  New tsel:                                    [0x01020301]
  Aliases:                                   COTS_SERVER COTS_server
```

Note: This menu may be accessed directly, using the command: **smit xtiservchg_sel**

Allows to display or modify the attributes relative to a Service.

The name of the service to be displayed or modified is selected in the list, and the corresponding attributes are displayed to be read or modified.

Remove a Service

```
 Remove a Service

 Name of the existing service:                          +
```

Note: This menu may be accessed directly, using the command: **smit xtiservrem_sel**

Removes a Service from the list of defined Services.

XTI onto NetShare Configurator

Access

Running the command:

smit xtinetshare

Description

This menu allows access to the configuration of XTI onto NetShare. Select the line corresponding to the type of parameters to be configured.

```
XTI onto NetShare

  XTI/TCP/IP hosts
  XTI/OSI services
```

Note: This sub-menu is available only if NetShare (RFC 1006) is installed.

As NetShare (RFC 1006) uses:

- the Internet addressing domain for Host machine declaration,
- OSI Transport selectors for Service definitions.

This sub-menu provides access to the two configuration tasks:

- XTI TCP/IP Hosts Configuration, on page 3-4,
- XTI OSI Services Configuration, on page 3-14.

XTI onto X.25 (XX25) Configurator

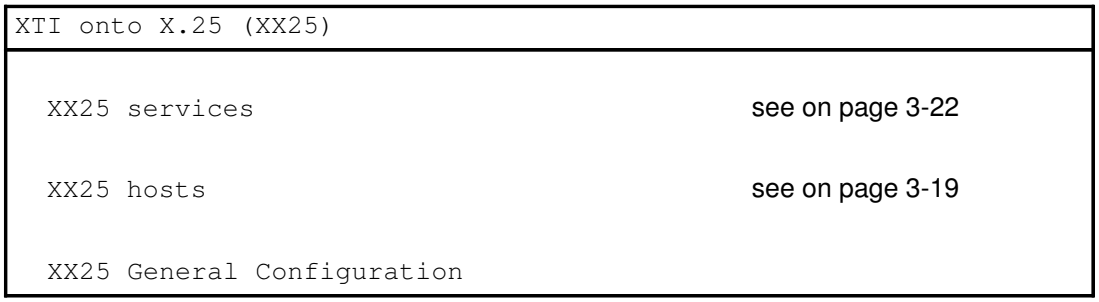
Access

Running the command:

smit xtix25

Description

This menu allows access to the configuration of XTI onto X.25. Select the line corresponding to the type of parameters to be configured.



Note: XX25 General Configuration is described in *HiSpeed WAN Comm. Installation and Services Guide*.

It may be accessed directly using **smit x25d**.

How to Manage XX25 Hosts

Access

Using the XTI onto X.25 configurator, run the command:

smit xtix25

Then select the function:

```
XX25 hosts
```

Description

An **XX25 Host** object defines a path within the network to access a remote host. The description of an **XX25 Host** includes the:

- remote **Host** name,
- **Virtual Circuit** type (listed as **Network Type** in the menus),
- **Remote Address**, i.e. the address used to access the remote network layer,
- **Local Address**, i.e. the address through which the connection goes out to the remote host.

The XX25 Hosts are saved in */etc/xtihosts* file and may be accessed using the XTI Name Server library.

This menu allows access to the configuration operations of XX25 hosts. Select the line corresponding to the action to be performed.

XX25 Hosts	
List All Hosts	see on page 3-20
Add a Host	see on page 3-20
Change / Show Characteristics of a Host	see on page 3-21
Remove a Host	see on page 3-21

Note: This menu may be accessed directly, using the command: **smit xx25host**

Associated Commands

xtihost

List All Hosts

```
Host        Remote address   Local address    Network Type  Lsa  Aliases
localhost    138002            138001           SVC             old_nt_1 example_1
lb01         138002            138001           SVC             0_to_1
lb01         138001            138002           SVC             0_to_1
hx25                           138001/1         PVC
h_x25_xx25   13800101          138002           SVC
```

Lists all the defined XX25 Hosts with their attributes, according to the displayed format:

- **Host** is the remote Host name.
- **Remote Address** (if any) is the X121 address of the Host.
- **Local Address** (if any) is the local X.121 address plus in case of PVC, the logical channel identifier.
- **Network Type** identifies the type of the Virtual Circuit used. It may be: SVC or PVC
- **Aliases**: are alternative names for *Host* (maximum 2).

Add a Host

```
 Add a Host

* Host name:                              []
* Provider name:                         npid_x25_cons
  Local  address:                         []                  +
  Remote address:                        []
* Circuit type:                           [SVC]               +
  Aliases:                               []
```

Note: This menu may be accessed directly, using the command: **smit xx25hostadd**

Allows all attributes, relative to a new Host to be added, to be defined.

Change/Show Characteristics of a Host

```
 Change / Show Characteristics of a Host

  Name of the existing host:                                          +

* Old host name:                        localhost
  New host name:                        [localhost]
* Provider name:                        npid_x25_cons
  New local  address:                    [138001]                    +
  New remote address:                    [138002]                    +
* Circuit type:                          [SVC]                       +
  Aliases:                              old_nt_1 example_1
```

Note: This menu may be accessed directly, using the command: **smit xx25hostchg_sel**

Allows the attributes relative to a Host to be displayed or modified.

The name of the host to be displayed or modified is selected from the list and the corresponding attributes to be read or modified are displayed.

Remove a Host

```
 Remove a Host

 Name of the existing host:                                           +
```

Note: This menu may be accessed directly, using the command: **smit xx25hostrem_sel**

Removes a Host from the list of defined Hosts.

How to Manage XX25 Services

Access

Using the XTI onto X.25 configurator, run the command:

smit xtix25

Then select the function:

```
XX25 services
```

Description

An **XX25 Service** object defines an association between a Server-Application Name (and aliases) and the Subsequent Application Identifier (SAI) to be used in order to access to this application from the network.

It must be defined both by the server which provides it as well as by the client who uses it.

The XX25 Services are saved in */etc/xtiservices* file and may be accessed using the XTI Name Server library.

The XX25 Services, which are part of the XTI product, are defined in the XTI DataBase automatically at the XTI installation:

cons_server, *cots_server*, *select_server*, *poll_server*, *benchd* and *tconnectd*.

Afterwards, new XX25 Services (user applications) can be defined in the XTI DataBase.

This menu allows access to the configuration operations of XX25 services. Select the line corresponding to the action to be performed.

XX25 services	
List All Services	see page 3-23
Add a Service	see page 3-23
Change / Show Characteristics of a Service	see page 3-24
Remove a Service	see page 3-24

Note: This menu may be accessed directly, using the command: **smit xx25serv**

Associated Commands

xtiserv

List All Services

```
 List All Services
Service          TSEL-SAI/Proto              Aliases

cons_server     0xC4020301/npid_x25_cons     CONS_SERVER CONS_server
cots_server     0xC4020302/npid_x25_cons     COTS_SERVER COTS_server
select_server   0xC4020303/npid_x25_cons     SELECT_SERVER SELECT_server
poll_server     0xC4020304/npid_x25_cons     POLL_SERVER POLL_server
benchd          0xC4020305/npid_x25_cons     BENCHD xtibenchd
tconnectd       0xC4020306/npid_x25_cons     TCONNECTD xtitconnectd
```

Lists all the defined XX25 Services with their attributes, according to the displayed format:

- **Service** is the Service name,
- **SAI** is the Subsequent Application Identifier (analogous to an OSI Transport Selector),
- **Protocol** is the Protocol name, as defined in the XTI Database (*/etc/xtiprotocols*)
  Always equal to *npid_x25_cons*
- **Aliases** are nicknames for Service name (maximum 2).

The example lists the services defined at the XTI/XX25 installation

Add a Service

```
 Add a Service

* Name:                                          []
* XX25 Subsequent Application Identifier (SAI):  []
* Protocol:                                      npid_x25_cons
  Aliases:                                       []
```

Note: This menu may be accessed directly, using the command: **smit xx25servadd**

Allows all attributes, relative to a new Service to be added, to be defined.

Change/Show Characteristics of a Service

```
Change / Show Characteristics of a Service

 Name of the existing service:                                    +

* Old name:                                      benchd
* Protocol:                                      npid_x25_cons
  New name:                                      [benchd]
  New SAI:                                       [0xC4020304]
  Aliases:                                       BENCHD xtibenchd
```

Note: This menu may be accessed directly, using the command: **smit xx25servchg_sel**

Allows attributes relative to a Service to be displayed or modified.

The name of the service to be displayed or modified is selected from the list and the corresponding attributes to be read or modified are displayed.

Remove a Service

```
 Remove a Service

  Name of the existing service:                                   +
```

Note: This menu may be accessed directly, using the command: **smit xx25servrem_sel**

Removes a Service from the list of defined Services.

XTI Option Profile Configurator

Access

Running the command:

smit xtiopt

Description

XTI Option Manager	
List profile(s)	see on page 3-26
Add a profile/option	see on page 3-26
Change characteristics of a profile/option	see on page 3-27
Remove a profile	see on page 3-27

An **Option Profile** object is a set of XTI options conformant to the format definition in *X/Open Transport Interface XPG4 CAE Specification Version 2*. An **Option Profile** is made of a set of items (*level*, *name*, *value*), where:

- *level* identifies the XTI level or a protocol of the transport provider, for instance INET_TCP or X25_NP,
- *name* identifies the option within the level,
- *value* is a value (optional) for the option.

Refer to **Appendix C. Options** for a complete list of the XTI Options.

The Option Profiles are saved in the */etc/xtiopts* file and may be accessed using the XTI Name Server library.

This menu allows access to the configuration of XTI Option Profiles. Select the line corresponding to the action to be performed.

Associated Commands

xtiopt

List an Option Profile

```
 List profile
 Profile to list:                          []                  +

PROFILE Name :
        OptionLevel  OptionName       Value(s)
PROFILE example_ltpdu :
        ISO_TP       TCO_LTPDU        1024
```

Note: This menu may be accessed directly, using the command: **smit xtioptlist**

Displays the characteristics of an option profile chosen from the list of defined Option Profiles.

Where:

- **ProfileName** is the name of the Option Profile. It is a string of 40 digits maximum.
- **OptionLevel** defines on which level the option is significant:

  XTI_GENERIC, INET_TCP, INET_UDP or INET_IP, ISO_TP, or X25_NP.
- **OptionName** defines the option according to the OptionLevel.
- **Value(s)** is a list of 40 digits maximum. Each value is separated by a comma.

Add an Option Profile

```
 Add a profile/option

* Profile Name:                        []                  +
* Level of option:                     []                  +
* Name of option:                      []                  +
  Value(s):                            []                  +
```

Note: This menu may be accessed directly, using the command: **smit xtioptadd**

Allows a new option profile to be defined or to add an option in an existing profile. The parameters to be defined are described in **List an Option Profile**. Except for the **ProfileName** which is user-specific, the values of the other parameters may be defined by choice from a list.

Change Characteristics of an Option Profile

```
Change characteristics of a profile/option
* Name of the Profile:                                          +

* Old profile name:                    example_ltpdu
  New profile name:                   [example_ltpdu]
  Option name:                          []                      +
  New level:                            []                      +
  New option:                           []                      +
  New value(s):                         []                      +
```

Note: This menu may be accessed directly, using the command: **smit xtioptchg_sel**

Allows an Option Profile to be modified.

The name of the Profile to be modified is selected from the list. For each option of the profile the parameters to be modified may be re-defined by selection from a list.

Remove an Option Profile

```
Remove a profile
* Name:                                 []                      +
```

Note: This menu may be accessed directly, using the command: **smit xtioptrem**

Removes a Profile from the list of defined Profiles.

XTI Trace Configurator

Access

Running the command:

smit xtitrace

Description

This menu allows access to the configuration of XTI Traces. Select the line corresponding to the type of operation to be performed.

XTI Trace	
Change/Show User Trace Levels	see on page 3-33
Use XTI Trace Utilities	see on page 3-36
Use Common Trace Utilities	
Change/Show Administrative Trace Levels	see on page 3-29

Note: Common Trace Utilities permits to trace any application or process, knowing the corresponding hook-id. This is equivalent to the command **smit trace**.

Refer to *AIX Performance Tuning Guide* for more details.

Refer to How to Use XTI Traces, on page 7-5, for the description of the impact of these XTI trace levels and for an example of XTI trace report.

How to Set XTI Administrative Trace Levels

Access

Using the XTI onto OSI configurator, run the command:

smit xtitrace

Then select the function:

```
Change/Show Administrative Trace Levels
```

Description

The Administrative Trace Levels may be modified only by the administrator (**root** authority) and are trace levels used by default if the user does not set specific trace levels.

Traces may be set in user-space (XTI Libraries) and/or in kernel-space (XTI kernel).

This menu allows access to the configuration of XTI administrative trace levels. Select the line corresponding to the action to be performed.

```
 Change/Show Administrative Trace Levels

Change/Show Default XTI Libraries Trace Levels    see on page 3-30

Change/Show Default XTI Libraries and Kernel Trace Levels
                                                  see on page 3-32

Change/Show XTI Kernel Trace Levels               see on page 3-31
```

Note: This menu may be accessed directly, using the command: **smit xtitraceglob**

Associated Command

xtitracelevel

Note: The trace levels are taken into account by an XTI application on a call to a *t_open()* or a *t_sync()* XTI function, or a call to a Name Server library function.

Set XTI Libraries Trace Levels

```
Change/Show Default XTI Libraries Trace Level

 Warning and protocol errors:                  yes
 CONNECTION functionnalities:                  yes       +
 MANAGEMENT functionnalities:                  yes       +
 DATA TRANSFER functionnalities:               yes       +
 Entry and return of external XTI lib. func.:  yes       +
 Description of I/O parameters values:         no        +
 States transitions in automatas:              no        +
 Data part of messages (limit 4096 bytes):     no        +
```

Note: This menu may be accessed directly, using the command: **smit xtitracegloblib**

Note: Default values are provided and displayed. All these trace levels may be set to **yes** or **no**, except for the **Warning and Protocol Errors** level which is always set to **yes**.

Sets the default trace levels in user-space (XTI libraries).

The resulting values are saved in **/etc/xtitrace** file.

The XTI Libraries trace levels have the following meaning:

- **Warning and protocol errors**: **XTI_LEVEL0**
- **CONNECTION functionalities: XTI_LEVEL28**

  If **Entry and return of external XTI lib. func.** is set, allows to trace the connection functions: t_accept, t_bind, t_close, t_connect, t_listen, t_open, t_rcvconnect, t_rcvdis, t_rcvrel, t_snddis, t_sndrel, t_unbind.

  If the t_listen function is executed in asynchronous mode and is the only XTI primitive called in a loop, the identical sequence of traces are not repeated. A trace indicates the number of repetitions.

- **MANAGEMENT functionalities**: **XTI_LEVEL29**

  If **Entry and return of external XTI lib. func.** is set, allows the management functions to be traced: t_alloc, t_error, t_free, t_getinfo, t_getstate, t_look, t_optmgmt, t_sync.

  If the t_look function is the only XTI primitive called in a loop, the identical sequence of traces is not repeated. A trace indicates the number of repetitions.

- **DATA TRANSFER functionalities**: **XTI_LEVEL30**

  If **Entry and return of external XTI lib. func.** is set, allows to trace the data transfer functions: t_rcv, t_rcvudata, t_rcvuderr, t_snd, t_sndudata.

- **Entry and return of external XTI lib. func.**: **XTI_LEVEL11**

  Validates the trace of XTI functions, according to the levels set previously, **CONNECTION**, **MANAGEMENT** or **DATA TRANSFER**.

  – On entry, trace reports the pointer addresses and simple parameter values given on the function call.

  – On exit, trace reports the return value if any.

- **Description of I/O parameters values**: **XTI_LEVEL24**

  This level must be set in conjunction with **Entry and return of external XTI lib. func.**

Allows input and output parameters values on the entry and exit of XTI functions to be traced, according to the levels set previously, **CONNECTION**, **MANAGEMENT** or **DATA TRANSFER**.

- **States transitions in automatas**: **XTI_LEVEL10**
- **Data part of messages (limit 4096 bytes)**: **XTI_LEVEL27**

  Allows the Data part of messages, transmitted through XTI, to be traced.

Set XTI Kernel Trace Levels

```
Change/Show XTI Kernel Trace Levels

Warning and protocol errors:                  yes
Description of I/O parameters values:         yes       +
States transitions in automatas:              yes       +
XTI kernel msg to (from) Provider Interface:  yes       +
Data part of messages (limit 4096 bytes):     no        +
```

Note: This menu may be accessed directly, using the command: **smit xtitraceglobxti3**

Note: Default values are provided and displayed. All these trace levels may be set to **yes** or **no**, except for the **Warning and Protocol Errors** level which is always set to **yes**.

Sets the default trace levels in kernel-space.

The XTI kernel trace levels have the following meaning:

- **Warning and protocol errors**: **XTI_LEVEL0**
- **Description of I/O parameters values**: **XTI_LEVEL24**

  Allows input and output parameter values to be traced on the entry and exit of XTI kernel STREAMS entry points: *open* ( ), *close* ( ), *wput* ( ) and *rput* ( ).
- **States transitions in automatas**: **XTI_LEVEL10**
- **XTI4 Kernel msg to (from) Provider Interface**: **XTI_LEVEL26**

  Allows Provider Interface messages sent by the **xti4mod** streams module to the lower layer to be traced and received by **xti4mod** from the lower layer.
- **Data part of messages (limit 4096 bytes)**: **XTI_LEVEL27**

  Allows the Data part of messages transmitted through XTI kernel part to be traced.

Set XTI Libraries and Kernel Trace Levels

```
Change/Show Default XTI Libraries and Kernel Trace Levels

 Warning and protocol errors:                    yes
 CONNECTION functionnalities:                    yes       +
 MANAGEMENT functionnalities:                    yes       +
 DATA TRANSFER functionnalities:                 yes       +
 Entry and return of all external XTI func.:     no        +
 Description of I/O parameters values:           no        +
 States transitions in automatas:                no        +
 XTI kernel msg to (from) Provider Interface:    no        +
 Data part of messages (limit 4096 bytes):       no        +
```

Note: This menu may be accessed directly, using the command: **smit xtitraceglobcnx**

Note: Default values are provided and displayed. All these trace levels may be set to **yes** or **no**, except for the **Warning and Protocol Errors** level which is always set to **yes**.

Sets the default trace levels in user-space and kernel-space.

The resulting values are saved in **/etc/xticnxtrace** file.

The XTI Libraries and Kernel trace levels have the same meaning as for XTI Libraries trace levels, on page 3-30 and for XTI Kernel trace levels, on page 3-31.

How to Set XTI User Trace Levels

Access

Using the XTI onto OSI configurator, run the command:

smit xtitrace

Then select the function:

```
Change/Show User Trace Levels
```

Description

This menu allows access to the configuration of XTI user trace levels. Select the line corresponding to the action to be performed.

Change/Show User Trace Levels	
Change/Show XTI Libraries Trace Levels	see on page 3-34
Change/Show XTI Libraries and Kernel Trace Levels	see on page 3-35

Note: This menu may be accessed directly, using the command: **smit xtitraceuser**

Any user may modify specific Trace Levels in user-space (XTI Libraries) and in kernel-space (XTI kernel). These user trace levels are saved in user files whose full path name are defined in shell environment variables **XTI_FILE_TRACE_LEVEL** and **XTI_FILE_TRACE_LEVELCNX**.

If these Shell environment variables **XTI_FILE_TRACE_LEVEL** and **XTI_FILE_TRACE_LEVELCNX** do not exist, the default trace levels defined by the administrator are used.

Associated Command

xtitracelevel

Note: The trace levels are taken into account by an XTI application on a call to a *t_open()* or a *t_sync()* XTI function, or a call to a Name Server library function.

Set XTI Libraries Trace Levels

```
Change/Show XTI Libraries Trace Level

  XTI libraries trace levels file:              []        +

 Warning and protocol errors:                   yes
 CONNECTION functionnalities:                   no        +
 MANAGEMENT functionnalities:                   no        +
 DATA TRANSFER functionnalities:                no        +
 Entry and return of external XTI lib. func.:   no        +
 Description of I/O parameters values:          no        +
 States transitions in automatas:               no        +
 Data part of messages (limit 4096 bytes):      no        +
```

Note: This menu may be accessed directly, using the command: **smit xtitraceuserlib**

Note: Default values are provided and displayed. All these trace levels may be set to **yes** or **no**, except for the **Warning and Protocol Errors** level which is always set to **yes**.

Sets the user trace levels in user-space (XTI libraries).

XTI libraries trace levels file is the file where the user library trace level are saved. This file is built with the same syntax as /**etc**/**xtitrace** file and is defined in the shell environment variable **XTI_FILE_TRACE_LEVEL**.

The user XTI Libraries trace levels have the same meaning as for Administrative XTI Libraries trace levels, on page 3-30.

Set XTI Libraries and Kernel Trace Levels

```
Change/Show XTI Libraries and Kernel Trace Levels

  XTI libraries and Kernel trace levels file:   []        +

 Warning and protocol errors:                   yes
 CONNECTION functionnalities:                   yes       +
 MANAGEMENT functionnalities:                   yes       +
 DATA TRANSFER functionnalities:                yes       +
 Entry and return of all external XTI func.:    no        +
 Description of I/O parameters values:          no        +
 States transitions in automatas:               no        +
 XTI kernel msg to (from) Provider Interface:   no        +
 Data part of messages (limit 4096 bytes):      no        +
```

Note: This menu may be accessed directly, using the command: **smit xtitraceusercnx**

Note: Default values are provided and displayed. All these trace levels may be set to **yes** or **no**, except for the **Warning and Protocol Errors** level which is always set to **yes**.

Sets the default trace levels in user-space and kernel-space.

XTI libraries and kernel trace levels file is the file where the user library and kernel trace levels are saved.
This file is built with the same syntax as /**etc/xticnxtrace** file and is defined in the shell environment variable **XTI_FILE_TRACE_LEVELCNX**.

The XTI Libraries and Kernel trace levels have the same meaning as for Administrative XTI Libraries trace levels, on page 3-30. An additional level is provided:

- **XTI4 Kernel msg to (from) Provider Interface**: **XTI_LEVEL26**

  Allows Provider Interface messages sent by the **xti4mod** streams module to the lower layer to be traced and received by **xti4mod** from the lower layer.

How to Use XTI Trace Utilities

Access

Using the XTI onto OSI configurator, run the command:

smit xtitrace

Then select the function:

```
Use XTI Trace Utilities
```

Description

This menu allows access to the configuration of XTI administrative trace levels. Select the line corresponding to the action to be performed.

```
Use XTI Trace Utilities

Start XTI Trace
Stop XTI Trace
Generate an XTI Trace Report
```

Note: This menu may be accessed directly, using the command: **smit xtitraceuse**

Associated Commands

trace, trcstop, trcrpt

Note: The commands **trace**, **trcstop** and **trcrpt** are described in *AIX Performance Tuning Guide*

Any user may access trace utilities in his own environment to:

- Start XTI Trace

  Equivalent to running AIX trace command in asynchronous mode with the hook–id 906 (XTI–API).

Note: This command fails if XTI trace has been started previously by another user.

- Stop XTI Trace
- Generate an XTI Trace Report on a user-specific trace file. By default the file /**tmp**/**xti.tr** is used.

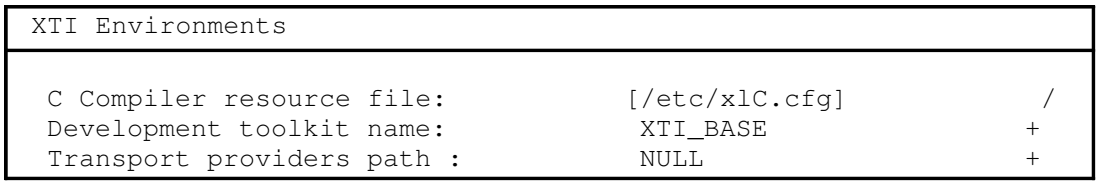
XTI Environments Configurator

Access

Running the command:

smit chxti

Description



This menu displays and allows the current XTI attribute to be modified:

- **C Compiler resource file** is the application development toolkit configuration file. The **/etc/xlC.cfg** is used by default, but any C compiler resource configuration file may be used instead.
- **Development toolkit name** defines which XTI library is used by default to compile an XTI application. (By default means using the standard **/etc/xlC.cfg** file and the **cc** command without any options)

  Three possible values may be chosen from the list:

  XTI–ENHANCED: for the Bull-enhanced XTI library over TCP/IP, OSI Transport and NetShare,

  XX25: for the XX25 library (Bull-enhanced XTI onto X.25),

  XTI–BASE for the AIX-issued XTI library.
- **Transport provider path** is a list of transport providers classified in priority order. This list is used for the automatic selection of providers by the **t_gettp()**(on page 5-15) function of the XTI Name Server. The transport providers and associated priority may be chosen from a list.

Associated Commands

lsxti	Displays the current XTI attributes.
chxti	Changes the current XTI attributes.

Chapter 4. XTI Library Functions

Each XTI function is described in conformity with *X/Open Transport Interface XPG4 CAE Specification Version 2*. The information specific to Transport Providers has been integrated in the function description and the information specific to Bull-enhanced XTI added under the headings:

- **TCP/IP Implementation Specifics**
- **OSI Implementation Specifics**
- **XX25 Implementation Specifics**
- **Bull Implementation Specifics**

For each XTI function, a table is given which summarizes the contents of the input and output parameter. The key is given below:

x	The parameter value is meaningful. (Input parameter must be set before the call and output parameter may be read after the call.)
(x)	The content of the object pointed to by the x pointer is meaningful.
?	The parameter value is meaningful but the parameter is optional.
(?)	The content of the object pointed to by the ? pointer is optional.
/	The parameter value is meaningless.
=	The parameter after the call keeps the same value as before the call.

Refer to How to Prepare a Bull-enhanced XTI Application, on page 7-2, to use the appropriate options in compiling and linking the application-program.

List of Bull-enhanced XTI Library Functions

t_accept Subroutine

Purpose

Accept a connect request.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
resfd	x	/
call–>addr.maxlen	/	/
call–>addr.len	x	/
call–>addr.buf	?(?)	/
call–>opt.maxlen	/	/
call–>opt.len	x	/
call–>opt.buf	?(?)	/
call–>udata.maxlen	/	/
call–>udata.len	x	/
call–>udata.buf	?(?)	/
call–>sequence	x	/

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

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

In *call*, *addr* is the protocol address of the calling transport user, *opt* indicates any options associated with the connection, *udata* points to any user data to be returned to the caller, and *sequence* is the value returned by *t_listen*() that uniquely associates the response with a previously received connect indication. The address of the caller, *addr* may be null (length zero). Where *addr* is not null then it may optionally be checked by XTI.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connect indication arrived. Before the connection can be accepted on the same endpoint (*resfd*==*fd*), the user must have responded to any previous connect indications received on that transport endpoint (via *t_accept*() or *t_snddis*()). Otherwise, *t_accept*() will fail and set *t_errno* to [TINDOUT].

If a different transport endpoint is specified (*resfd*!=*fd*), then the user may or may not choose to bind the endpoint before the t_accept is issued. If the endpoint is not bound prior to the

t_accept(), then the transport provider will automatically bind it to the same protocol address *fd* is bound to. If the transport user chooses to bind the endpoint it must be bound to a protocol address with a *qlen* of zero and must be in the T_IDLE state before the *t_accept*() is issued.

The call to *t_accept*() will fail with *t_errno* set to [TLOOK] if there are indications (e.g. connect or disconnect) waiting to be received on the endpoint *fd*.

The *udata* argument enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider as returned in the *connect* field of the *info* argument of *t_open*() or *t_getinfo*(). If the *len* field of *udata* is zero, no data will be sent to the caller. All the *maxlen* fields are meaningless.

When the user does not indicate any option (call–>opt.len = 0) it is assumed that the connection is to be accepted unconditionally. The transport provider may choose options other than the defaults to ensure that the connection is accepted successfully.

Caveats

There may be transport provider specific restrictions on address binding.

Some transport providers do not differentiate between a connect indication and the connection itself. If the connection has already been established after a successful return or *t_listen*(), *t_accept*() will assign the existing connection to the transport endpoint specified by *resfd*.

Refer to **OSI Implementation Specifics** and **TCP/IP Implementation Specifics** for more information.

Valid States

fd: T_INCON

resfd (fd!=resfd): T_IDLE

Errors

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

[TBADF] The file descriptor *fd* or *resfd* does not refer to a transport endpoint.

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

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

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

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

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

[TBADSEQ] An invalid sequence number was specified.

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

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

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

[TINDOUT] The function was called with *fd*==*resfd* but there are outstanding connection indications on the endpoint. Those other connection indications must be

handled either by rejecting them via *t_snddis*(3) or by accepting them on a different endpoint via *t_accept*(3).

[TPROVMISMATCH]
The file descriptors *fd* and *resfd* do not refer to the same transport provider.

[TRESQLEN] The endpoint referenced by *resfd* (where *resfd* != *fd*) was bound to a protocol address with a *qlen* that is greater than zero.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

[TRESADDR] This transport provider requires both *fd* and *resfd* to be bound to the same address. This error results if they are not.

Return Values

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

TCP/IP Implementation Specifics

Issuing *t_accept( )* assigns an already established connection to *resfd*.

Since user data cannot be exchanged during the connection establishment phase, *call–>udata.len* must be set to 0. Also, *resfd* must be bound to the same address as *fd*. A potential restriction on the binding of endpoints to protocol addresses is described under *t_bind( )*.

If association related options (IP_OPTIONS, IP_TOS) are to be sent with the connect confirmation, the values of these options must be set with *t_optmgmt( )* before the T_LISTEN event occurs. When the transport user detects a T_LISTEN, TCP has already established the connection. Association-related options passed with *t_accept( )* become effective at once, but since the connection is already established, they are transmitted with subsequent IP diagrams sent out in the T_DATAXFER state.

OSI Implementation Specifics

The parameter *call–>udata.len* must be in the range 0 to 32. The user may send up to 32 octets of data when accepting the connection.

If *fd* is not equal to *resfd*, *resfd* should either be in the state T_UNBND or be in the state T_IDLE and be bound to the same address as *fd* with the *qlen* parameter set to 0.

A process can listen for an incoming indication on a given *fd* and then accept the connection on another endpoint *resfd* which has been bound to the same or a different protocol address with the *qlen* parameter (of the *t_bind( )* function) set to 0. The protocol address bound to the new accepting endpoint (*resfd*) should in general be the same as the listening endpoint (fd), because at the present time, the ISO transport service definition (ISO 8072:1986) does not authorise acceptance of an incoming connection indication with a responding address different from the called address, except under certain conditions (see ISO 8072:1986 paragraph 12.2.4, Responding Address), but it also states that it may be changed in the future.

The t_accept subroutine is not applicable for ConnectionLess Transport Service.

XX25 Implementation Specifics

The *call–>udata.len* parameter must be:

- 0 in basic format,
- in the range 0 to 128 in extended format, format negotiated using the option T_X25_FASTSELECT. (Refer to Appendix C. Bull-enhanced XTI Options.)

Bull Implementation Specifics

The options (*call–>opt* parameter) supported by Bull-enhanced XTI, XTI_GENERIC, ISO_TP, INET_TCP and INET_IP, X25_NP, are listed in Bull-enhanced XTI Options in Appendix C.

See also

t connect(), *t getstate*(), *t listen*(), *t open*(), *t optmgmt*(), *t rcvconnect*().

t_alloc Subroutine

Purpose

Allocate a library structure.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
struct_type	x	/
fields	x	/

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

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

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

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

Each of the above structures, except T_INFO, contains at least one field of type **struct netbuf.** For each field of this type, the user may specify that the buffer for that field should be allocated as well. The length of the buffer allocated will be equal to or greater than the appropriate size as returned in the *info* argument of *t_open*() or *t_getinfo*(). The relevant fields of the *info* argument are described in the following list. The *fields* argument specifies which buffers to allocate, where the argument is the bitwise-or of any of the following:

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

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

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

T_ALL — All relevant fields of the given structure. Fields which are not supported by the transport provider specified by *fd* will not be allocated.

For each relevant field specified in *fields*, *t_alloc*() will allocate memory for the buffer associated with the field, and initialise the *len* field to zero and the *buf* pointer and *maxlen* field accordingly. Irrelevant or unknown values passed in fields are ignored. Since the length of the buffer allocated will be based on the same size information that is returned to the user on a call to *t_open*() and *t_getinfo*(), *fd* must refer to the transport endpoint through which the newly allocated structure will be passed. In this way the appropriate size information can be accessed. If the size value associated with any specified field is –1 or –2 (see *t_open*()

or *t_getinfo*()), *t_alloc*() will be unable to determine the size of the buffer to allocate and will fail, setting *t_errno* to [TSYSERR] and *errno* to [EINVAL]. For any field not specified in *fields*, *buf* will be set to the null pointer and *len* and *maxlen* will be set to zero.

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

Valid States

ALL – apart from T_UNINIT.

Errors

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

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

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

[TNOSTRUCTYPE]
Unsupported *struct_type* requested. This can include a request for a structure type which is inconsistent with the transport provider type specified, that is, connection–oriented or connectionless.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

On successful completion, *t alloc*() returns a pointer to the newly allocated structure. On failure, a null pointer is returned.

See also

t_free(), *t_getinfo*(), *t_open*().

t_bind Subroutine

Purpose

Bind an address to a transport endpoint.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
req–>addr.maxlen	/	/
req–>addr.len	x >=0	/
req–>addr.buf	x(x)	/
req–>qlen	x >=0	/
ret–>addr.maxlen	x	/
ret–>addr.len	/	x
ret–>addr.buf	?	(?)
ret–>qlen	/	x >=0

This function associates a protocol address with the transport endpoint specified by *fd* and activates that transport endpoint. In connection mode, the transport provider may begin enqueuing incoming connect indications or servicing a connection request on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

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

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

The *addr* field of the **t_bind** structure specifies a protocol address, and the *qlen* field is used to indicate the maximum number of outstanding connect indications.

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

If the requested address is not available, *t_bind*() will return –1 with *t_errno* set as appropriate. If no address is specified in *req* (the *len* field of *addr* in *req* is zero or *req* is NULL) the transport provider will assign an appropriate address to be bound, and will return that address in the *addr* field of *ret*. If the transport provider could not allocate an address, *t_bind*() will fail with *t_errno* set to [TNOADDR].

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

The *qlen* field has meaning only when initialising a connection–mode service. It specifies the number of outstanding connect indications that the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider but which has not been accepted or rejected. A value of *qlen* greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of *qlen* will be negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications. However, this value of *qlen* will never be negotiated from a requested value greater than zero to zero. This is a requirement on transport providers; see **CAVEATS** below. On return, the *qlen* field in *ret* will contain the negotiated value.

If *fd* refers to a connection-mode service, this function allows more than one transport endpoint to be bound to the same protocol address (however, the transport provider must also support this capability), but it is not possible to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address. In other words, only one *t_bind()* for a given protocol address may specify a value of *qlen* greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of *qlen* greater than zero, *t_bind()* will return –1 and set *t_errno* to [TADDRBUSY]. When a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of the connection, until a *t_unbind()* or *t_close()* call has been issued. No other transport endpoints may be bound for listening on that same protocol address while that initial listening endpoint is active (in the data transfer phase or in the T_IDLE state). This will prevent more than one transport endpoint bound to the same protocol address from accepting connect indications.

If *fd* refers to a connectionless-mode service, only one endpoint may be associated with a protocol address. If a user attempts to bind a second transport endpoint ot an already bound protocol address, *t_bind()* will return –1 and set *t_errno* to [TADDRBUSY].

Valid States

T_UNBND

Errors

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

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

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

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

[TNOADDR] The transport provider could not allocate an address.

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

[TBUFOVFLW] The number of bytes allowed for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The provider's state will change to T_IDLE and the information to be returned in *ret* will be discarded.

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

[TADDRBUSY] The requested address is in use.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

Caveats

The requirement that the value of *qlen* never be negotiated from a requested value greater than zero to zero implies that transport providers, rather than the XTI implementation itself, accept this restriction.

A transport provider may not allow an explicit binding of more than one transport endpoint to the same protocol address, although it allows more than one connection to be accepted for the same protocol address. To ensure portability, it is, therefore, recommended not to bind transport endpoints that are used as responding endpoints (*resfd*) in a call to *t_accept*(), if the responding address is to be the same as the called address.

TCP/IP Implementation Specifics

The *addr* field of the **t_bind** structure represents the local socket, i.e. an address which specifically includes a port identifier.

In the connection-oriented mode (i.e. TCP), the *t_bind( )* function may only bind one transport endpoint to any particular protocol address. If that endpoint was bound in passive mode, i.e. *qlen* > 0, then other endpoints will be bound to the passive endpoint's protocol address via the *t_accept*( ) function only; that is, if *fd* refers to the passive endpoint and *resfd* refers to the new endpoint on which the connection is to be accepted, *resfd* will be bound to the same protocol address as *fd* after the successful completion of the *t_accept*( ) function.

OSI Implementation Specifics

The *addr* field of the *t_bind* structure represents the local TSAP.

XX25 Implementation Specifics

The address field of the *t_bind()* structure contains the matching requirements for routing incoming calls to the endpoint. This may include (but is not limited to) representations of one or more of the following:

- a local SNPA identifier,
- a local X.25 address,
- a local X.25 subaddress,
- a local NSAP,
- a call user data matching requirement,
- a PVC number.

Where an incoming call can be routed to multiple endpoints on basis of their matching requirements, the actual endpoint selected will be implementation dependent.

Note: An implementation may choose to provide support for a wildcard mechanism for address information, for exemple to route incoming calls whose call user data starts with a particular pattern.

Bull Implementation Specifics

OSI Addressing

The OSI Communication Stack addressing is implementation specific and is described in **Appendix D. OSI Addressing**

XX25 Addressing

The XX25 addressing is implementation specific and is described in **Appendix E. XX25 Addressing**

See also

t_alloc(), *t_close*(), *t_open*(), *t_optmgmt*(), *t_unbind*().

t_close Subroutine

Purpose

Close a transport endpoint.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/

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

The function *t_close()* should be called from the T_UNBND state (see *t_getstate()*). However, this function does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, *close()* will be issued for that file descriptor; the *close()* will be abortive if there are no other descriptors in this, or in another process which reference the transport endpoint, and in this case will break any transport connection that may be associated with that endpoint.

A *t_close()* issued on a connection endpoint may cause data previously sent, or data not yet received, to be lost. It is the responsibility of the transport user to ensure that data is received by the remote peer.

Valid States

ALL – apart from T_UNINIT.

Errors

On failure, *t_errno* is set to the following:

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

See also

t_getstate(), *t_open*(), *t_unbind*().

t_connect Subroutine

Purpose

Establish a connection with another transport user.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
sndcall–>addr.maxlen	/	/
sndcall–>addr.len	x	/
sndcall–>addr.buf	x(x)	/
sndcall–>opt.maxlen	/	/
sndcall–>opt.len	x	/
sndcall–>opt.buf	x(x)	/
sndcall–>udata.maxlen	/	/
sndcall–>udata.len	x	/
sndcall–>udata.buf	?(?)	/
sndcall–>sequence	/	/
rcvcall–>addr.maxlen	x	/
rcvcall–>addr.len	/	x
rcvcall–>addr.buf	?	(?)
rcvcall–>opt.maxlen	x	/
rcvcall–>opt.len	/	x
rcvcall–>opt.buf	?	(?)
rcvcall–>udata.maxlen	x	/
rcvcall–>udata.len	/	x
rcvcall–>udata.buf	?	(?)
rcvcall–>sequence	/	/

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

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

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

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

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

The *opt* argument permits users to define the options that may be passed to the transport provider. These options are specific to the underlying protocol of the transport provider and are listed for ISO and TCP protocols in Bull-enhanced XTI Options in Appendix C. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

If used, *sndcall–>opt.buf* must point to a buffer with the corresponding options; the *maxlen* and *buf* fields of the **netbuf** structure pointed by *rcvcall–>addr* and *rcvcall–>opt* must be set before the call.

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

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

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

When a synchronous *t_connect()* call is interrupted by the arrival of a signal, the state of the corresponding transport endpoint is T_OUTCON, allowing a further call to either *t_rcvconnect*(), *t_rcvdis*(), or *t_snddis*().

Valid States

T_IDLE

Errors

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

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

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

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

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

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

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

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

[TBUFOVFLW] The number of bytes allowed for an incoming argument is greater than 0 but not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the information to be returned in *rcvcall* will be discarded.

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

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

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

[TADDRBUSY] This transport provider does not support multiple connections with the same local and remote addresses. This error indicates that a connection already exists.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

TCP/IP Implementation Specifics

The *sndcall–>addr* structure specifies the remote socket. In the present version, the returned address set in *rcvcall–>addr* will have the same value. Since user data cannot be exchanged during the connection establishment phase, *sndcall–>udata.len* must be set to 0.

Note that the peer TCP, and not the peer transport user, confirms the connection.

OSI Implementation Specifics

The *sndcall–>addr* structure specifies the remote called TSAP. In the present version, the returned address set in *rcvcall–>addr* will have the same value.

The setting of *sndcall–>udata* is optional for ISO connections, but with no data, the *len* field of *udata* must be set to 0. The *maxlen* and *buf* fields of the **netbuf** structure, pointed to by *rcvcall–>addr* and *rcvcall–>opt*, must be set before the call.

The t_connect subroutine is not applicable for ConnectionLess Transport Service.

XX25 Implementation Specifics

The *sndcall–>udata.len* parameter must be:

- in the range 0 to 16 in basic format,
- in the range 0 to 128 in extended format, format negotiated using the option T_X25_FASTSELECT. (Refer to Appendix C. Bull-enhanced XTI Options.)

The *sndcall–>addr* is used to select either an SVC or a PVC.

- For an SVC the *sndcall–>addr* structure contains a representation of the addressing information necessary to reach the destination, it may contain (but is not limited to) one or more of the following:
  – a remote X.25 address,
  – a local X.25 address,
  – a call user data matching requirement.
  When the connection has been established, the *rcvcall–>addr* structure represents the address on which the call has been accepted.
- For a PVC, the *sndcall–>addr* structure contains the PVC number to be used (*t_connect()* associates the user with the PVC). If it is already in use, the error [TADDRBUSY] is returned. On successful return:
  – In synchronous mode, the PVC will be in state **T_DATAXFER**,
  – In asynchronous mode, the PVC will be in state **T_OUTCON** and a **T_CONNECT** event will be outstanding.
  When the connection has been established, the *rcvcall–>addr* structure represents the actual PVC allocated.

Bull Implementation Specifics

OSI Addressing

The OSI Communication Stack addressing is implementation specific and is described in **Appendix D. OSI Addressing**

NetShare (RFC 1006) Addressing

The same behavior as for the pure OSI Communication Stack and a subset of the wildcard values (**OSI_TSEL_ANY**, **OSI_NSAP_ANY**) are provided by the NetShare (RFC 1006) provider.

The other specific values defined for pure OSI Communication Stack are meaningless for the NetShare (RFC 1006) provider.

Figure 3. NetShare (RFC 1006) Addressing

XX25 Addressing

The XX25 addressing is implementation specific and is described in **Appendix E. XX25 Addressing**.

Supported Options:

The supported options are listed in Bull-enhanced XTI Option Profiles in Appendix C.

See also

t_accept(), *t_alloc*(), *t_getinfo*(), *t_listen*(), *t_open*(), *t_optmgmt*(), *t_rcvconnect*().

t_error Subroutine

Purpose

Produce error message.

Syntax

#include <xti.h>

int t_error (*errmsg***)**
char **errmsg***;**

Description

Parameters	Before call	After call
errmsg	x	/

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

The error message is written as follows: first (if *errmsg* is not a null pointer and the character pointed to be *errmsg* is not the null character) the string pointed to by *errmsg* followed by a colon and a space; then a standard error message string for the current error defined in *t_errno*. If *t_errno* has a value different from [TSYSERR], the standard error message string is followed by a newline character. If, however, *t_errno* is equal to [TSYSERR], the *t_errno* string is followed by the standard error message string for the current error defined in *errno* followed by a newline.

The language for error message strings written by *t_error*() is implementation–defined. If it is in English, the error message string describing the value in *t_errno* is identical to the comments following the *t_errno* codes defined in **xti.h**. The contents of the error message strings describing the value in *errno* are the same as those returned by the *strerror(3C)* function with an argument of *errno*.

The error number, *t_errno*, is only set when an error occurs and it is not cleared on successful calls.

Example

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

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

The diagnostic message to be printed would look like:

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

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

Valid States

ALL– apart from T_UNINIT.

Errors

No errors are defined for the *t_error()* function.

Return Values

Upon successful completion, a value of 0 is returned.

t_free Subroutine

Purpose

Free a library structure.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
ptr	x	/
struct_type	x	/

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

The argument *ptr* points to one of the seven structure types described for *t_alloc*(), and *struct_type* identifies the type of that structure which must be one of the following:

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

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

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

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

Valid States

ALL – apart from T_UNINIT.

Errors

On failure, *t_errno* is set to the following:

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

[TNOSTRUCTYPE] Unsupported *struct_type* requested.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

See also

t_alloc()

t_getinfo Subroutine

Purpose

Get protocol–specific service information.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
info–>addr	/	x
info–>options	/	x
info–>tsdu	/	x
info–>etsdu	/	x
info–>connect	/	x
info–>discon	/	x
info–>servtype	/	x
info–>flags	/	x

This function returns the current characteristics of the underlying transport protocol and/or transport connection associated with file descriptor *fd*. The *info* pointer is used to return the same information returned by *t_open*(), although not necessarily precisely the same values. This function enables a transport user to access this information during any phase of communication.

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

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

The values of the fields have the following meanings:

addr A value greater than zero indicates the maximum size of a transport protocol address and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than zero indicates the maximum number of bytes of protocol–specific options supported by the provider and a value of –2 specifies that the transport provider does not support user–settable options.

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

etsdu A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider. Note that the semantics of expedited data might be quite different for different transport providers. Refer to **OSI Implementation Specifics** and **TCP/IP Implementation Specifics**.

connect A value greater than zero specifies the maximum amount of data that may be associated with connection establishment functions and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than zero specifies the maximum amount of data that may be associated with the *t_snddis*() and *t_rcvdis*() functions and a value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

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

flags This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates that the underlying transport provider supports the sending of zero-length TSDUs.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the *t_alloc*() function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of option negotiation during connection establishment (the *t_optmgmt*() call has no effect on the values returned by *t_getinfo*()). These values will only change from the values presented to *t_open*() after the endpoint enters the T_DATAXFER state.

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

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

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

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

Valid States

ALL – apart from T_UNINIT.

Errors

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

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

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

TCP/IP Implementation Specifics

This table is specific to Bull implementation.

Parameters	Before call	After call	
		TCP/IP	UDP/IP
fd	x	/	/
info–>addr		16	16
info–>options	/	512	512
info–>tsdu	/	0	8192
info–>etsdu	/	–1	–2
info–>connect	/	–2	–2
info–>discon	/	–2	–2
info–>servtype	/	T_COTS/T_COTS_ORD	T_CLTS
info–>flags	/	T_SNDZERO	T_SNDZERO

OSI Implementation Specifics

The information returned by *t_getinfo*() reflects the characteristics of the transport connection or, if no connection is established, the maximum characteristics a transport connection could take on using the underlying transport provider. In all possible states except T_DATAXFER, the function *t_getinfo*() returns in the parameter *info* the same information as was returned by *t_open()*. In T_DATAXFER, however, the information returned may differ from that returned by *t_open*(), depending on:

- the transport class negotiated during connection establishment (ISO transport provider only), and
- the negotiation of expedited data transfer for this connection.

In T_DATAXFER, the *estdu* field in the **t_info** structure is set to –2 if no expedited data transfer was negotiated, and to 16 otherwise. The remaining fields are set according to the characteristics of the transport protocol class in use for this connection, as defined in the table below, specific to Bull implementation.

Parameters	Before call	After call		
		Connection class 0	Connection class 1–4	ISO–over–TCP
fd	x	/	/	/
info–>addr		107	107	68
info–>options	/	2000	2000	256
info–>tsdu	/	–1	–1	–1
info–>etsdu	/	–2	16/–2 (1)	16
info–>connect	/	–2	32	32
info–>discon	/	–2	64	–2
info–>servtype	/	T_COTS	T_COTS	T_COTS
info–>flags	/	0	0	0

1. Depending on the negotiation of expedited data transfer.

XX25 Implementation Specifics

The information returned by *t_getinfo()* reflects the characteristics of the X.25 connection or, if no connection is established, the maximum characteristics an X.25 connection could take on using the underlying X.25 provider.

The parameters of the *t_getinfo()* function, for the different versions of the X.25 protocol ((X.25–1980, X.25–1984, X.25–1988, X.25–1993, and so on) are presented in the table below.

Parameters	Before call	After call	
		X.25–1988 X.25–1984	X.25–1980
fd	x	/	/
info–>addr		x	x
info–>options	/	x	x
info–>tsdu	/	x (1)	x (1)
info–>etsdu	/	–2/32 (2)	–2/1 (3)
info–>connect	/	16/128 (4)	16/128 (4)
info–>discon	/	0/128 (5)	0/128 (5)
info–>servtype	/	T_COTS	T_COTS
info–>flags	/	T_SENDZERO	T_SENDZERO

1. –1 or an integral number greater than zero.
2. –2 if no expedited data transfer can be exchanged, and 32 otherwise.
3. –2 if no expedited data transfer can be exchanged, and 1 otherwise.
4. 16 in basic format or 128 in extended format (if the T_X25_FASTSELECT option has been negotiated).
5. 0 in basic format or 128 in extended format (if the T_X25_FASTSELECT option has been negotiated).

See also

t_alloc(), *t_open*().

t_getprotaddr Subroutine

Purpose

Get the protocol address.

Syntax

#include <xti.h>

int t_getprotaddr (*fd, boundaddr, peeraddr***)**
int *fd***;**
struct t_bind **boundaddr***;*
struct t_bind **peeraddr***;*

Description

Parameters	Before call	After call
fd	x	/
boundaddr–>maxlen	x	/
boundaddr–>addr.len	/	x
boundaddr–>addr.buf	?	(?)
boundaddr–>qlen	/	/
peeraddr–>maxlen	x	/
peeraddr–>addr.len	/	x
peeraddr–>addr.buf	?	(?)
peeraddr–>qlen	/	/

The *t_getprotaddr()* function returns local and remote protocol addresses currently associated with the transport endpoint specified by *fd*. In *boundaddr* and *peeraddr*, the user specifies *maxlen*, which is the maximum size of the address buffer and *buf* which points to the buffer where the address is to be placed. On return, the *buf* field of *boundaddr* points to the address, if any, currently bound to *fd*, and the *len* field specifies the length of the address. If the transport endpoint is in the T_UNBND state, zero is returned in the *len* field of *boundaddr*. The *buf* field of the *peeraddr* points to the address, if any, currently connected to *fd*, and the *len* field specifies the length of the address. If the transport endpoint is not in the T_DATAXFER state, zero is returned in the *len* field of *peeraddr*.

Valid States

ALL – apart from T_UNINIT.

Errors

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

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

[TBUFOVFLW] The number of bytes allowed for an incoming argument is greater than 0 but not sufficient to store the value of that argument.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

See also

t_bind().

t_getstate Subroutine

Purpose

Get the current state.

Syntax

#include <xti.h>

int t_getstate (*fd***)**
int *fd***;**

Description

Parameters	Before call	After call
fd	x	/

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

Valid States

ALL – apart from T_UNINIT.

Errors

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

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

[TSTATECHNG] The transport provider is undergoing a transient state change.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

State is returned upon successful completion. Otherwise, a value of –1 is returned and *t_errno* is set to indicate an error. The current state is one of the following:

T_UNBND unbound

T_IDLE idle

T_OUTCON outgoing connection pending

T_INCON incoming connection pending

T_DATAXFER data transfer

T_OUTREL outgoing orderly release (waiting for an orderly release indication)

T_INREL incoming orderly release (waiting to send an orderly release request)

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

See also

t_open().

t_listen Subroutine

Purpose

Listen for a connect indication.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
call–>addr.maxlen	x	/
call–>addr.len	/	x
call–>addr.buf	?	(?)
call–>opt.maxlen	x	/
call–>opt.len	/	x
call–>opt.buf	?	(?)
call–>udata.maxlen	x	/
call–>udata.len	/	x
call–>udata.buf	?	(?)
call–>sequence	/	x

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

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

In *call*, *addr* returns the protocol address of the calling transport user. This address is in a format usable in future calls to *t_connect*(). Note, however, that *t_connect*() may fail for other reasons, for example [T_ADDRBUSY]. *opt* returns options associated with the connect request, *udata* returns any user data sent by the caller on the connect request and *sequence* is a number that uniquely identifies the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

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

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

Valid States

T_IDLE, T_INCON.

Errors

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

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

[TBADQLEN] The argument *qlen* of the endpoint referenced by *fd* is zero.

[TBUFOVFLW] The number of bytes allocated for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect indication information to be returned in *call* is discarded. The value of *sequence* returned can be used to do a *t snddis*().

[TNODATA] O_NONBLOCK was set, but no connect indications had been queued.

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

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

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

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

[TQFULL] The maximum number of outstanding indications has been reached for the endpoint referenced by *fd*.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Caveats

Some transport providers do not differenciate between a connect indication and the connection itself. If this is the case, a successful return of t_listen() indicates an existing connection.

TCP/IP Implementation Specifics

Upon successful return, *t_listen()* indicates an existing connection and not a connection indication.

Since user data cannot be exchanged during the connection establishment phase, *call–>udata.maxlen* must be set to zero before the call to *t_listen()*. The *call–>addr* structure contains the remote calling socket.

OSI Implementation Specifics

The *call–>addr* structure contains the remote calling TSAP. Since, at most, 32 octets of data will be returned with the connect indication, *call–>udata.maxlen* should be set to 32 before the call to *t_listen*().

If the user has set *qlen* greater than 1 (on the call to *t_bind*()), then the user may queue up several connect indications before responding to any of them. The user should be forewarned that the ISO transport provider may start a timer to be sure of obtaining a response to the connect request in a finite time. So if the user queues the connect indications for too long before responding to them, the transport provider initiating the connection will disconnect it.

Bull Implementation Specifics

OSI Addressing

The OSI Communication Stack addressing is implementation specific and is described in **Appendix D. OSI Addressing**

XX25 Addressing

The XX25 addressing is implementation specific and is described in **Appendix E. XX25 Addressing**

Supported Options

The supported options are listed in **Appendix C.** Bull-enhanced XTI Option Profiles.

Return Values

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

See also

fcntl(), *t_accept*(), *t_alloc*(), *t_bind*(), *t_connect*(), *t_open*(), *t_optmgmt*(), *t_rcvconnect*().

t_look Subroutine

Purpose

Look at the current event on a transport endpoint.

Syntax

#include <xti.h>

int t_look (*fd***)**
int *fd***;**

Description

Parameters	Before call	After call
fd	x	/

This function returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is calling functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, [TLOOK], on the current or next function to be executed. Details on events which cause functions to fail [T_LOOK] may be found in *X/Open Transport Interface XPG4 CAE Specification Version 2*,
Section 4.6, Events and TLOOK Error Indication

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

Valid States

ALL – apart from T_UNINIT.

Errors

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

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

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

T_LISTEN connection indication received.

T_CONNECT connect confirmation received.

T_DATA normal data received.

T_EXDATA expedited data received.

T_DISCONNECT
disconnect received.

T_UDERR datagram error indication.

T_ORDREL orderly release indication.

T_GODATA Flow control restrictions on normal data flow that led to a [TFLOW] error have been lifted. Normal data may be sent again.

T_GOEXDATA Flow control restrictions on expedited data flow that led to a [TFLOW] error have been lifted. Expedited data may be sent again.

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

TCP/IP Implementation Specifics

As soon as a segment with the TCP urgent pointer set enters the TCP receive buffer, the event T_EXDATA is indicated. T_EXDATA remains set until all data up to the byte pointed to by the TCP urgent pointer has been received. If the urgent pointer is updated, and the user has not yet received the byte previously pointed to by the urgent pointer, the update is invisible to the user.

See also

t_open(), *t_snd*(), *t_sndudata*().

Application Usage

Additional functionality is provided through the Event Management interface (EM).

t_open Subroutine

Purpose

Establish a transport endpoint.

Syntax

#include <xti.h>
#include <fcntl.h>

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

Description

Parameters	Before call	After call
name	x	/
oflag	x	/
info–>addr	/	x
info–>options	/	x
info–>tsdu	/	x
info–>etsdu	/	x
info–>connect	/	x
info–>discon	/	x
info–>servtype	/	x
info–>flags	/	x

The *t_open*() function must be called as the first step in the initialisation of a transport endpoint. This function establishes a transport endpoint by supplying a transport provider identifier that indicates a particular transport provider (i.e., transport protocol) and returning a file descriptor that identifies that endpoint.

The argument *name* points to a transport provider identifier and *oflag* identifies any open flags (as in *open*()). The argument *oflag* is constructed from O_RDWR optionally bitwise inclusive–or'ed with O_NONBLOCK. These flags are defined by the header **<fcntl.h>**. The file descriptor returned by *t_open*() will be used by all subsequent functions to identify the particular local transport endpoint.

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

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

```
long discon;    /* max amount of data allowed on */
                /* t_snddis() and t_rcvdis() functions */
long servtype;  /* service type supported by the */
                /* transport provider */
long flags;     /* other info about the transport provider */
```

The values of the fields have the following meanings:

addr A value greater than zero indicates the maximum size of a transport protocol address and a value of –2 specifies that the transport provider does not provide user access to transport protocol addresses.

options A value greater than zero indicates the maximum number of bytes of protocol–specific options supported by the provider and a value of –2 specifies that the transport provider does not support user–settable options.

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

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

connect A value greater than zero specifies the maximum amount of data that may be associated with connection establishment functions and a value of –2 specifies that the transport provider does not allow data to be sent with connection establishment functions.

discon A value greater than zero specifies the maximum amount of data that may be associated with the *t_snddis*() and *t_rcvdis*() functions and a value of –2 specifies that the transport provider does not allow data to be sent with the abortive release functions.

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

flags This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates the underlying transport provider supports the sending of zero–length TSDUs.

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

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

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

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

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

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

If *info* is set to a null pointer by the transport user, no protocol information is returned by *t_open*().

Valid States

T_UNINIT.

Errors

On failure, *t_errno* is set to the following:

[TBADFLAG] An invalid flag is specified.

[TBADNAME] Invalid transport provider name.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

A valid file descriptor is returned upon successful completion. Otherwise, a value of –1 is returned and *t_errno* is set to indicate an error.

TCP/IP Implementation Specifics

t_open() is called as the first step in the initialisation of a transport endpoint. This function returns various default characteristics of the underlying transport protocol by setting fields in the **t_info** structure.

The following, specific to Bull implementation, should be the values returned by the call to *t_open*() with the indicated transport providers.

Parameters	Before call	After call	
		TCP/IP	UDP/IP
name	x	/	/
oflag	x	/	/
info–>addr	/	16	16
info–>options	/	512	512
info–>tsdu	/	0	8192
info–>etsdu	/	–1	–2
info–>connect	/	–2	–2
info–>discon	/	–2	–2
info–>servtype	/	T_COTS/T_COTS_ORD	T_CLTS
info–>flags	/	T_SENDZERO	T_SENDZERO

OSI Implementation Specifics

The function *t_open* is called as the first step in the initialisation of a transport endpoint. This function returns various default characteristics associated with the different classes. According to ISO 8073:1992, an OSI transport provider supports one or several out of five different transport protocols, class 0 through class 4. The default characteristics returned in the parameter *info* are those of the highest-numbered protocol class the transport provider is able to support. If, for example, a transport provider supports classes 2 and 0, the characteristics returned are those of class 2.

The table below, specific to Bull implementation, gives the characteristics associated with the different classes.

Parameters	Before call	After call			
		Connection class 0	Connection class 1–4	ISO–over–TCP	CLTP
name	x	/	/	/	/
oflag	x	/	/	/	/
info–>addr	/	107	107	68	107
info–>options	/	2000	2000	256	2000
info–>tsdu	/	–1	–1	–1	(1)
info–>etsdu	/	–2	16	16	–2
info–>connect	/	–2	32	32	–2
info–>discon	/	–2	64	–2	–2
info–>servtype	/	T_COTS	T_COTS	T_COTS	T_CLTS
info–>flags	/	0	0	0	0

1. Depending on the Profile network (size of NSDU).

XX25 Implementation Specifics

The function *t_open()* is called at the first step in the initialisation of an X.25 endpoint. This function returns various default characteristics associated with the different versions of X.25 that are supported. If, for example an X.25 provider supports X.25–1984 and X.25–1988, the characteristics returned are those of X.25–1988. Il the X.25 provider is limited to X.25–1980, the characteristics returned are those of X.25–1980.

The parameters of the *t_open()* function, for the different versions of the X.25 protocol (X.25–1980, X.25–1984, X.25–1988, X.25–1993, and so on) are presented in the table below.

Parameters	Before call	After call	
		X.25–1988 X.25–1984	X.25–1980
fd	x	/	/
info–>addr		x	x
info–>options	/	x	x
info–>tsdu	/	x (1)	x (1)
info–>etsdu	/	–2/32 (2)	–2/1 (3)
info–>connect	/	16/128 (4)	16/128 (4)
info–>discon	/	0/128 (5)	0/128 (5)
info–>servtype	/	T_COTS	T_COTS
info–>flags	/	T_SENDZERO	T_SENDZERO

1. –1 or an integral number greater than zero.
2. –2 if no expedited data transfer can be exchanged, and 32 otherwise.
3. –2 if no expedited data transfer can be exchanged, and 1 otherwise.
4. 16 in basic format or 128 in extented format (if the X.25 facility *Fast Select* has been negotiated).

5. 0 in basic format or 128 in extended format (if the X.25 facility *Fast Select* has been negotiated).

See also

open().

t_optmgmt Subroutine

Purpose

Manage options for a transport endpoint.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
req–>opt.maxlen	/	/
req–>opt.len	x	/
req–>opt.buf	x (x)	/
req–>flags	x	/
ret–>opt.maxlen	x	/
ret–>opt.len	/	x
ret–>opt.buf	?	(?)
ret–>flags	/	x

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

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

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

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

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

Each option in the options buffer is defined in a **t_opthdr** structure possibly followed by an option value.

```
struct t_opthdr {
        unsigned long len;
                /* total option length- sizeof(struct t_opthdr)
                   + length of option value in bytes */
        unsigned long level;    /* protocol affected */
        unsigned long name;     /* option name */
        unsigned long status;   /* status value */
        /* followed by the option value */
};
```

The *level* field of **struct t_opthdr** identifies the XTI level or a protocol of the transport provider. The *name* field identifies the option within the level, and *len* contains its total length, i.e. the length of the option header **t_opthdr** plus the length of the option value. If *t_optmgmt*() is called with the action T_NEGOTIATE set, the *status* field of the returned options contains information about the success or failure of a negotiation.

Each option in the input or output option buffer must start at a long–word boundary. The macro **OPT_NEXTHDR(pbuf, buflen, poption)** can be used for that purpose. The parameter *pbuf* denotes a pointer to an option buffer *opt.buf*, and *buflen* is its length. The parameter *poption* points to the current option in the option buffer. **OPT_NEXTHDR** returns a pointer to the position of the next option or returns a null pointer if the option buffer is exhausted. The macro is helpful for writing and reading. Refer to **<xti.h>** include file for the exact definition.

If the transport user specifies several options on input, all options must address the same level.

If any option in the options buffer does not indicate the same level as the first option, or the level specified is unsupported, then the *t_optmgmt*() request will fail with [TBADOPT]. If the error is detected, some options have possibly been successfully negotiated. The transport user can check the current status by calling *t_optmgmt*() with the T_CURRENT flag set.

Note: In *X/Open Transport Interface XPG4 CAE Specification Version 2* (Chapter 5. **The Use of Options**), a detailed description about the use of options should be read before using this function.

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

T_NEGOTIATE This action enables the transport user to negotiate option values.
The user specifies the options of interest and their values in the buffer specified by *req–>opt.buf* and *req–>opt.len*. The negotiated option values are returned in the buffer pointed to by *ret–>opt.buf*. The *status* field of each returned option is set to indicate the result of the negotiation. The value is T_SUCCESS if the proposed value was negotiated, T_PARTSUCCESS if a degraded value was negotiated, T_FAILURE if the negotiation failed (according to the negotiation rules), T_NOTSUPPORT if the transport provider does not support this option or illegally requests negotiation of a privileged option, and T_READONLY if modification of a read–only option was requested. If the status is T_SUCCESS, T_FAILURE, T_NOTSUPPORT or T_READONLY, the returned option value is the same as the one requested on input.
The overall result of the negotiation is returned in *ret–>flags*.
This field contains the worst single result, whereby the rating is done according to the order T_NOTSUPPORT, T_READONLY, T_FAILURE, T_PARTSUCCESS, T_SUCCESS. The value T_NOTSUPPORT is the worst result and T_SUCCESS is the best.

For each level, the option T_ALLOPT (see below) can be requested on input. No value is given with this option; only the **t_opthdr** part is specified. This input requests to negotiate all supported options of this level to their default values. The result is returned option by option in *ret–>opt.buf*. (Note that depending on the state of the transport endpoint, not all requests to negotiate the default value may be successful.)

T_CHECK This action enables the user to verify whether the options specified in *req* are supported by the transport provider.
If an option is specified with no option value (it consists only of a **t_opthdr** structure), the option is returned with its *status* field set to T_SUCCESS if it is supported, T_NOTSUPPORT if it is not or needs additional user privileges, and T_READONLY if it is read–only (in the current XTI state). No option value is returned.
If an option is specified with an option value, the *status* field of the returned option has the same value, as if the user had tried to negotiate this value with T_NEGOTIATE. If the status is T_SUCCESS, T_FAILURE, T_NOTSUPPORT or T_READONLY, the returned option value is the same as the one requested on input.
The overall result of the option checks is returned in *ret–>flags*. This field contains the worst single result of the option checks, whereby the rating is the same as for T_NEGOTIATE.
Note that no negotiation takes place. All currently effective option values remain unchanged.

T_DEFAULT This action enables the transport user to retrieve the default option values. The user specifies the options of interest in *req–>opt.buf*. The option values are irrelevant and will be ignored; it is sufficient to specify the **t_opthdr** part of an option only. The default values are then returned in *ret–>opt.buf*.
The *status* field returned is T_NOTSUPPORT if the protocol level does not support this option or the transport user illegally requested a privileged option, T_READONLY if the option is read–only, and set to T_SUCCESS in all other cases. The overall result of the request is returned in *ret–>flags*. This field contains the worst single result, whereby the rating is the same as for T_NEGOTIATE.
For each level, the option T_ALLOPT (see below) can be requested on input. All supported options of this level with their default values are then returned. In this case, *ret–>opt.maxlen* must be given at least the value of *info–>options* (see *t_getinfo*(), *t_open*()) before the call.

T_CURRENT This action enables the transport user to retrieve the currently effective option values. The user specifies the options of interest in *req–>opt.buf*. The option values are irrelevant and will be ignored; it is sufficient to specify the **t_opthdr** part of an option only. The currently effective values are then returned in *ret–>opt.buf*.
The *status* field returned is T_NOTSUPPORT if the protocol level does not support this option or the transport user illegally requested a privileged option, T_READONLY if the option is read–only, and set to T_SUCCESS in all other cases. The overall result of the request is returned in *ret–>flags*. This field contains the worst single result, whereby the rating is the same as for T_NEGOTIATE.
For each level, the option T_ALLOPT (see below) can be requested on input. All supported options of this level with their currently effective values are then returned.

The option **T_ALLOPT** can only be used with *t_optmgmt*() and the actions T_NEGOTIATE, T_DEFAULT and T_CURRENT. It can be used with any supported level and addresses all supported options of this level. The option has no value; it consists of a **t_opthdr** only. Since in a *t_optmgmt*() call only options of one level may be addressed, this option should not be requested together with other options. The function returns as soon as this option has been processed.

Options are independently processed in the order they appear in the input option buffer. If an option is multiply input, it depends on the implementation whether it is multiply output or whether it is returned only once.

Transport providers may not be able to provide an interface capable of supporting T_NEGOTIATE and/or T_CHECK functionalities. When this is the case, the error [TNOTSUPPORT] is returned.

The function *t_optmgmt*() may block under various circumstances and depending on the implementation. The function will block for instance, if the protocol addressed by the call resides on a separate controller. It may also block due to flow control constraints, i.e. if data sent previously across this transport endpoint has not yet been fully processed. If the function is interrupted by a signal, the option negotiations that have been done so far may remain valid. The behaviour of the function is not changed if O_NONBLOCK is set.

XTI–level options

XTI–level options are not specific for a particular transport provider. An XTI implementation supports none, all or any subset of the options defined below. An implementation may restrict the use of any of these options by offering them only in the privileged or read–only mode, or if *fd* relates to specific transport providers.

The subsequent options are not association–related. They may be negotiated in all XTI states except T_UNINIT.

The protocol level is **XTI_GENERIC**. For this level, the following options are defined:

option name	type of option value	legal option value	meaning
XTI_DEBUG	array of unsigned longs	see text	enable debugging
XTI_LINGER	struct linger	see text	linger on close if data is present
XTI_RCVBUF	unsigned long	size in octets	receive buffer size
XTI_RCVLOWAT	unsigned long	size in octets	receive low–water mark
XTI_SNDBUF	unsigned long	size in octets	send buffer size
XTI_SNDLOWAT	unsigned long	size in octets	send low–water mark

Figure 4. XTI-level options

A request for XTI_DEBUG is an absolute requirement.

A request to activate XTI_LINGER is an absolute requirement; the timeout value to this option is not.

XTI_RCVBUF, XTI_RCVLOWAT, XTI_SNDBUF and XTI_SNDLOWAT are not absolute requirements.

XTI_DEBUG This option enables debugging. The values of this option are implementation–defined. Debugging is disabled if the option is specified with ”no value”, i.e. with an option header only.

The system supplies utilities to process the traces. Note that an implementation may also provide other means for debugging.

XTI_LINGER This option is used to linger the execution of a *t_close*() or *close*() if send data is still queued in the send buffer. The option value specifies the linger period. If a *close*() or *t_close*() is issued and the send buffer is not empty, the system attempts to send the pending data within the linger period before closing the endpoint. Data still pending after the linger period has elapsed is discarded.

Depending on the implementation, *t_close*() or *close*() either block for at maximum the linger period, or immediately return, whereupon the system holds the connection in existence for at most the linger period.

The option value consists of a structure **t_linger** declared as:

```
struct t_linger {
        long l_onoff;   /* switch option on/off */
        long l_linger;  /* linger period in seconds */
}
```

Legal values for the field *l_onoff* are:
T_NO switch option off
T_YES activate option.

The value *l_onoff* is an absolute requirement.

The field *l_linger* determines the linger period in seconds. The transport user can request the default value by setting the field to T_UNSPEC. The default timeout value depends on the underlying transport provider (it is often T_INFINITE) and all non-negative numbers.

The *l_linger* value is not an absolute requirement. The implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Note that this option does not linger the execution of *t_snddis*().

XTI_RCVBUF This option is used to adjust the internal buffer size allocated for the receive buffer. The buffer size may be increased for high-volume connections, or decreased to limit the possible backlog of incoming data.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_RCVLOWAT

This option is used to set a low–water mark in the receive buffer. The option value gives the minimum number of bytes that must have been accumulated in the receive buffer before they become visible to the transport user. If and when the amount of accumulated receive data exceeds the low–water mark, a T_DATA event is created, an event mechanism (e.g. *poll*() or *select*()) indicates the data, and the data can be read by *t_rcv*() or *t_rcvudata*().

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_SNDBUF This option is used to adjust the internal buffer size allocated for the send buffer.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_SNDLOWAT

This option is used to set a low–water mark in the send buffer. The option value gives the minimal number of bytes that must have accumulated in the send buffer before they are sent.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

Valid States

ALL – apart from T_UNINIT.

Errors

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

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

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

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

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

[TBADFLAG] An invalid flag was specified.

[TBUFOVFLW] The number of bytes allowed for an incoming argument (*maxlen*) is gretaer than 0 but not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

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

Return Values

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

Bull Implementation Specifics

Supported Options

The options supported by Bull-enhanced XTI: XTI_GENERIC, ISO_TP, INET_TCP and INET_IP, X25_NP are listed in Bull-enhanced XTI Options in Appendix C.

The XTI_DEBUG option of the XTI_GENERIC protocol level is implemented through the Bull-enhanced XTI **Trace tool**.

Setting Trace Levels

t_optmgmt() allows to modify XTI Trace Levels in a whole application or in some parts of the application. The XTI Trace Levels thus defined overwrite those previously configured by the administrator or the user.

```
struct t_opthdr {
        unsigned long len;
                /* total option length- sizeof(struct t_opthdr)
                   + length of option value in bytes */
        unsigned long level;    /* protocol affected */
        unsigned long name;     /* option name */
        unsigned long status;   /* status value */
        /* followed by the option value */
};

struct t_deblevel {
        unsigned long LEVEL_type;       /* CON_TRACE_LEVEL=1  */
        unsigned long LEVEL_value;      /* Trace level value */
};
```

The XTI trace levels are defined in a *t_opthdr* structure form, followed by a *t_deblevel* structure which provides the option value (see <xti.h>):

- the *level* field of *t_opthdr* equals XTI_GENERIC.
- the *name* field equals XTI_DEBUG.
- the *len* field equals the total lenght = size of *t_opthdr* + size of *t_deblevel.*
- the *LEVEL_type* field of *t_deblevel* equals CON_TRACE_LEVEL, which means that the traces are set both in the XTI library and XTI kernel.
- the *LEVEL_value* field contains the XTI trace levels.
  The Trace level variable is a 32 bits-variable, where each bit represents a trace level.
  The trace levels are a set of values between XTI_LEVEL0 and XTI_LEVEL32.
  XTI_LEVELn is tested with the mask 2^{n-1} (the n–th bit from the least significant bit) and XTI_LEVEL0 is always set.

XTI_LEVEL0	Warning and protocol errors.
XTI_LEVEL10	State transitions in Automatas. For each transition, the old state, the received event and the new state are traced.
XTI_LEVEL11	Entry and exit points of every XTI libraries functions and xti4mod module primitives (open, close and put procedures). An entry trace reports the pointers addresses and simple parameters values given on the function call and an exit trace reports the return value if any. In order to trace contents of input and output parameters, XTI_LEVEL24 must be set.
XTI_LEVEL24	Input and output parameters. Contents of input and output parameters are traced on the entry and exit of each function. Must be set with XTI_LEVEL11.
XTI_LEVEL26	TPI messages sent by the xti4mod module to the lower layer and those received xti4mod from the lower layer.
XTI_LEVEL27	Data part of messages transmitted through XTI.
XTI_LEVEL28	CONNECTION functionnalities. Trace the connection functions**,** t_accept(), t_bind(), t_close(), t_connect(), t_listen(), t_open(), t_rcvconnect(), t_rcvdis(), t_rcvrel(), t_snddis(), t_sndrel(), t_unbind(). Must be set with XTI_LEVEL11.

Note: If the **t_listen()** primitive is executed in asynchronous mode and is the only XTI primitive called in a loop, the identical sequence of traces are not repeated. A trace indicates the number of repetitions.

XTI_LEVEL29 MANAGEMENT functionnalities.
Trace the management functions**,** t_alloc()**,** t_error()**,** t_free()**,** t_getinfo()**,** t_getstate()**,** t_look()**,** t_optmgmt()**,** t_sync()**,** and t_optmgmt().
Must be set with XTI_LEVEL11.

Note: If the **t_look()** primitive is executed in asynchronous mode and the only XTI primitive called in a loop, the identical sequence of traces are not repeated. A trace indicates the number of repetitions.

XTI_LEVEL30 DATA TRANSFER functionnalities.
Trace the data transfer functions**,**t_rcv(), t_rcvudata(), t_rcvuderr(), t_snd() and t_sndudata()**.**
Must be set with XTI_LEVEL11.

See also

t_accept(), *t_alloc*(), *t_connect*(), *t_getinfo*(), *t_listen*(), *t_open*(), *t_rcvconnect*().

Bull-enhanced XTI Options in Appendix C.

t_rcv Subroutine

Purpose

Receive data or expedited data sent over a connection.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
buf	x	(x)
nbytes	x	/
flags	/	x

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

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

On return from the call, if T_MORE is set in *flags*, this indicates that there is more data and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple *t_rcv*() calls. In the asynchronous mode, the T_MORE flag may be set on return from the *t_rcv()* call even when the number of bytes received is less than the size of the receive buffer specified. Each *t_rcv*() with the T_MORE flag set indicates that another *t_rcv*() must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a *t_rcv*() call with the T_MORE flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from *t_open*() or *t_getinfo*(), the T_MORE flag is not meaningful and should be ignored. If *nbytes* is greater than zero on the call to *t_rcv*(), *t_rcv*() will return 0 only if the end of a TSDU is being returned to the user.

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

In synchronous mode, the only way for the user to be notified of the arrival of normal or expedited data is to issue this function or check for the T_DATA or T_EXDATA events using the *t_look* function. Additionally, the process can arrange to be notified via the EM interface.

Valid States

T_DATAXFER, T_OUTREL.

Errors

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

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

[TNODATA] O_NONBLOCK was set, but no data is currently available from the transport provider.

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

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

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

TCP/IP Implementation Specifics

The T_MORE flag should be ignored if normal data is delivered. If a byte in the data stream is pointed to by the TCP urgent pointer, as many bytes as possible preceding this marked byte and the marked byte itself are denoted as urgent data and are received with the T_EXPEDITED flag set. If the buffer supplied by the user is too small to hold all urgent data, the T_MORE flag will be set, indicating that urgent data still remains to be read. Note that the number of bytes received with the T_EXPEDITED flag set is not necessarily equal to the number of bytes sent by the peer user with the T_EXPEDITED flag set.

OSI Implementation Specifics

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

The t_rcv subroutine is not applicable for ConnectionLess Transport Service. See *t_rcvudata().*

XX25 Implementation Specifics

The behaviour of the function *t_rcv()* remains unchanged. The function can operate in synchronous and synchronous modes. It follows the current flow control rules.

The default behaviour is to acknowledge, in an automatic way, data sent with the Delivery Confirmation bit and expedited data.

The optional explicit acknowledgement is selected in the functions *t_optmgmt(), t_connect(), t_accept()* either with the **T_X25_USER_DACK** option, for the acknowledgement of data sent with the D bit, or with the **T_X25_USER_EACK** option, for the acknowledgement of expedited data.

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

In addition to the **T_EXPEDITED** and **T_MOR**E flags, the followings flags can be set in the argument *flags:*

- On return from the call, if T_X25_D is set in *flags*, this indicates that the data returned was sent with the D bit, and the **T_X25_USER_DACK** option is set. This data has to be acknowledged explicitly by the receiver.
- On return from the call, if the **T_X25_DACK** flag is set, data, previously sent with the D bit, has been acknowledged.
- On return from the call, if the **T_X25_EACK** flag is set, the previously sent expedited data has been acknowledged.

  Note: If either of **T_X25_DACK** or **T_X25_EACK** is set in *flags*, then no other flags are set and no user data is returned to the user.
- On return from the call, if **T_X25_Q** is set in *flags*, the data returned are qualified.
- On return from the call, if **T_X25_RST** is set in *flags*, this indicates that a reset indication occurred.

When **T_X25_RST** is returned, the argument *buf* contains the cause and diagnostic of the reset. Each one is coded into one octet. The cause is encoded in the first octet, and the diagnostic in the second octet. If the user's buffer is less than two bytes long then the diagnostic value is discarded, and if the length is zero the cause is also discarded.

See also

fcntl(), *t_getinfo*(), *t_look*(), *t_open*(), *t_snd*().

t_rcvconnect Subroutine

Purpose

Receive the confirmation from a connect request.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
call–>addr.maxlen	x	/
call–>addr.len	/	x
call–>addr.buf	?	(?)
call–>opt.maxlen	x	/
call–>opt.len	/	x
call–>opt.buf	?	(?)
call–>udata.maxlen	x	/
call–>udata.len	/	x
call–>udata.buf	?	(?)
call–>sequence	/	/

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

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

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

In *call*, *addr* returns the protocol address associated with the responding transport endpoint, *opt* presents any options associated with the connection, *udata* points to optional user data that may be returned by the destination transport user during connection establishment and *sequence* has no meaning for this function.

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

If O_NONBLOCK is set (via *t_open*() or *fcntl*()), *t_rcvconnect*() executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, *t_rcvconnect*() fails and returns immediately without waiting for the connection to be established. (See [TNODATA] below.) In this case, *t_rcvconnect*() must be called again to complete the connection establishment phase and retrieve the information returned in *call*.

Valid States

T_OUTCON

Errors

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

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

[TBUFOVFLW] The number of bytes allocated for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument and the connect information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to T_DATAXFER.

[TNODATA] O_NONBLOCK was set, but a connect confirmation has not yet arrived.

[TLOOK] An asynchronous event has occurred on this transport connection and requires immediate attention.

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

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

TCP/IP Implementation Specifics

Since user data cannot be exchanged during the connection establishment phase, *call–>udata.maxlen* must be set to 0 before the call to *t_rcvconnect()*. On return, the *call–>addr* structure contains the remote calling socket.

OSI Implementation Specifics

On return, the *call–>addr* structure contains the remote calling TSAP. Since, at most, 32 octets of data will be returned to the user, *call–>udata.maxlen* should be set to 32 before the call to *t_rcvconnect()*.

XX25 Implementation Specifics

On return:

- the *call–>addr* structure contains the responding X.25 address,
- the *call–>udata.len* field is
  - 0 in basic format,
  - in the range 0 to 128 in extended format, format negotiated using the option T_X25_FASTSELECT. (Refer to Appendix C. Bull-enhanced XTI Options.)

  Therefore *call–>udata.maxlen* should be set to 0 or 128, according to the format, before the call to *t_rcvconnect()*.

Bull Implementation Specifics

OSI Addressing

The OSI Communication Stack addressing is implementation specific and is described in **Appendix D. OSI Addressing**

XX25 Addressing

The XX25 addressing is implementation specific and is described in **Appendix E. XX25 Addressing**.

Supported Options

The supported options are listed in **Appendix C.** Bull-enhanced XTI Option Profiles.

See also

t_accept(), *t_alloc*(), *t_bind*(), *t_connect*(), *t_listen*(), *t_open*(), *t_optmgmt*().

t_rcvdis Subroutine

Purpose

Retrieve information from disconnect.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
discon–>udata.maxlen	x	/
discon–>udata.len	/	x
discon–>udata.buf	?	(?)
discon–>reason	/	x
discon–>sequence	/	?

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

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

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

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

Valid States

T_DATAXFER, T_OUTCON, T_OUTREL, T_INREL, T_INCON (ocnt >0).

Errors

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

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

[TNODIS] No disconnect indication currently exists on the specified transport endpoint.

[TBUFOVFLW] The number of bytes allocated for incoming data (*maxlen*) is greater than 0 but not sufficient to store the data. If *fd* is a passive endpoint with *ocnt* > 1, it remains in state T_INCON; otherwise, the endpoint state is set to T_IDLE.

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

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

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

TCP/IP Implementation Specifics

Since data may not be sent with a disconnect, the *discon–>udata* structure will not be meaningful.

OSI Implementation Specifics

Since at most 64 octets of data will be returned to the user, *discon–>udata.maxlen* should be set to 64 before the call to *t_rcvdis*().

XX25 Implementation Specifics

The field *discon–>reason* contains the X.25 cause and diagnostic of the connection release. The cause and the diagnostic are both encoded in an octet and can be retrieved by using respectively the **T_X25_GET_CAUSE** macro and the **T_X25_GET_DIAG** macro.

This function allows operations in accordance with XTI, but cannot be used to retrieve charging information or the address of the user that released the connection. For these purposes, the user has to call the function *t_optmgmt()* and retrieve the meaningful options.

On return, the *discon–>udata.len* field is

- 0 in basic format,
- in the range 0 to 128 in extended format, format negotiated using the option T_X25_FASTSELECT. (Refer to Appendix C. Bull-enhanced XTI Options.)

Therefore *discon–>udata.maxlen* should be set to 0 or 128, according to the format, before the call to *t_rcvdis()*.

Bull Implementation Specifics

The field *reason* specifies the reason for the disconnect and its meaning is protocol-dependent:

For TCP/IP

[XTID_TPINIT]	Initiated by transport provider.
[XTID_REMWITHDRAW]	Connect request withdrawn by remote.
[XTID_REMREJECT]	Connect request rejected by remote.
[XTID_REMINIT]	Disconnect request initiated by remote.

For OSI

The reason can be analyzed using the command **pmaderror.**

For XX25

Causes and diagnostics are described in ISO 8208 standard.

For NetShare (RFC 1006)

Same values as for TCP/IP and this additional reason.

0x80	Normal disconnect initiated by session entity.

See also

t_alloc(), *t_connect*(), *t_listen*(), *t_open*(), *t_snddis*().

t_rcvrel Subroutine

Purpose

Acknowledge receipt of an orderly release indication.

Syntax

#include <xti.h>

int t_rcvrel (*fd***)**
int *fd***;**

Description

Parameters	Before call	After call
fd	x	/

This function is used to acknowledge receipt of an orderly release indication. The argument *fd* identifies the local transport endpoint where the connection exists. After receipt of this indication, the user may not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if *t_sndrel*() has not been called by the user. This function is an optional service of the transport provider, and is only supported if the transport provider returned service type T_COTS_ORD on *t_open*() or *t_getinfo*().

Valid States

T_DATAXFER, T_OUTREL.

Errors

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

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

[TNOREL] No orderly release indication currently exists on the specified transport endpoint.

[TLOOK] An asynchronous event has occurred on this transport connection and requires immediate attention.

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

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

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

Uppon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *t_errno* is set to indicate an error.

See also

t_getinfo(), *t_open*(), *t_sndrel*().

t_rcvudata Subroutine

Purpose

Receive a data unit.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
unitdata–>addr.maxlen	x	/
unitdata–>addr.len	/	x
unitdata–>addr.buf	?	(?)
unitdata–>opt.maxlen	x	/
unitdata–>opt.len	/	x
unitdata–>opt.buf	?	(?)
unitdata–>udata.maxlen	x	/
unitdata–>udata.len	/	x
unitdata–>udata.buf	?	(?)
flags	/	x

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

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

The *maxlen* field of *addr*, *opt* and *udata* must be set before calling this function to indicate the maximum size of the buffer for each.

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

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

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

Valid States

T_IDLE.

Errors

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

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

[TNODATA] O_NONBLOCK was set, but no data units are currently available from the transport provider.

[TBUFOVFLW] The number of bytes allocated for the incoming protocol address or options (*maxlen*) is greater than 0 but not sufficient to store the information. The unit data information to be returned in *unitdata* will be discarded.

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

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

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

OSI Implementation Specifics

The *unitdata–>addr* structure specifies the remote TSAP. If the T_MORE flag is set, an additional *t_rcvudata()* call is needed to retrieve the entire TSDU. Only normal data is returned via the *t_rcvudata()* call. This function is not supported by an ISO–over–TCP transport provider.

Bull Implementation Specifics

Supported options

The supported options are listed in Bull-enhanced XTI Option Profiles in Appendix C.

See also

fcntl(), *t_alloc*(), *t_open*(), *t_rcvuderr*(), *t_sndudata*().

t_rcvuderr Subroutine

Purpose

Receive a unit data error indication.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
uderr–>addr.max-len	x	/
uderr–>addr.len	/	x
uderr–>addr.buf	?	(?)
uderr–>opt.maxlen	x	/
uderr–>opt.len	/	x
uderr–>opt.buf	?	(?)
uderr–>error	/	x

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

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

The *maxlen* field of *addr* and *opt* must be set before calling this function to indicate the maximum size of the buffer for each.

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

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

Valid States

T_IDLE.

Errors

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

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

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

[TBUFOVFLW] The number of bytes allocated for the incoming protocol address or options (*maxlen*) is greater than 0 but not sufficient to store the information. The unit data error information to be returned in *uderr* will be discarded.

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

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

OSI Implementation Specifics

The *uderr–>addr* structure contains the remote TSAP.

Bull Implementation Specifics

Supported options

The supported options are listed in Bull-enhanced XTI Option Profiles in Appendix C.

See also

t_rcvudata(), *t_sndudata*().

t_snd Subroutine

Purpose

Send data or expedited data over a connection.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
buf	x (x)	/
nbytes	x	/
flags	x	/

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

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

T_MORE If set in *flags*, this indicates to the transport provider that the transport service data unit (TSDU) (or expedited transport service data unit – ETSDU) is being sent through multiple *t_snd*() calls. Each *t_snd*() with the T_MORE flag set indicates that another *t_snd*() will follow with more data for the current TSDU (or ETSDU).

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

The sending of a zero–length fragment of a TSDU or ETSDU is only permitted where this is used to indicate the end of a TSDU or ETSDU, i.e. when the T_MORE flag is not set. Some transport providers also forbid zero–length TSDUs and ETSDUs. Refer to *X/Open Transport Interface XPG4 CAE Specification Version 2*, Appendix A, for a fuller explanation.

By default, *t snd*() operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NONBLOCK is set (via *t_open*() or *fcntl*()), *t_snd*() will execute in asynchronous mode, and will fail immediately if there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared via either *t_look*() or the EM interface.

On successful completion, *t_snd*() returns the number of bytes accepted by the transport provider. Normally this will equal the number of bytes specified in *nbytes*. However, if O_NONBLOCK is set, it is possible that only part of the data will actually be accepted by the transport provider. In this case, *t_snd*() will return a value that is less than the value of *nbytes*. If *nbytes* is zero and sending of zero octets is not supported by the underlying transport service, *t_snd*() will return –1 with *t_errno* set to [TBADDATA].

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as returned in the TSDU or ETSDU fields of the *info* argument returned by *t_getinfo*().

The error [TLOOK] may be returned to inform the process that an event (e.g., a disconnect) has occurred.

Valid States

T_DATAXFER, T_INREL.

Errors

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

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

[TBADDATA] Illegal amount of data:

- A single send was attempted specifying a TSDU (ETSDU) or fragment TSDU (ETSDU) greater than that specified by the current values of the TSDU or ETSDU fields in the *info* argument;
- a send of a zero byte TSDU (ETSDU) or zero byte fragment of a TSDU (ETSDU) is not supported by the provider,
- multiple sends were attempted resulting in a TSDU (ETSDU) larger than that specified by the current value of the TSDU or ETSDU fields in the *info* argument – the ability of an XTI implementation to detect such an error case is implementation–dependent (see **Caveats** below and **Implementation Specifics**).

[TBADFLAG] An invalid flag was specified.

[TFLOW] O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.

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

[TLOOK] An asynchronous event has occurred on this transport endpoint.

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

On successful completion, *t_snd*() returns the number of bytes accepted by the transport provider. Otherwise, –1 is returned on failure and *t_errno* is set to indicate the error.

Note: In asynchronous mode, if the number of bytes accepted by the transport provider is less than the number of bytes requested, this may indicate that the transport provider is blocked due to flow control.

Caveats

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. Therefore if several processes issue concurrent *t_snd*() calls then the different data may be intermixed.

Multiple sends which exceed the maximum TSDU or ETSDU size may not be discovered by XTI. In this case, an implementation–dependent error will result (generated by the transport provider) perhaps on a subsequent XTI call. This error may take the form of a connection abort, a [TSYSERR], a [TBADDATA] or a [TPROTO] error.

If multiple sends which exceed the maximum TSDU or ETSDU size are detected by XTI, *t_snd*() fails with [TBADDATA].

TCP/IP Implementation Specifics

The T_MORE flag should be ignored. If *t_snd()* is called with more than one byte specified and with the T_EXPEDITED flag set, then the last byte of the buffer will be the byte pointed to by the TCP urgent pointer. If the T_EXPEDITED flag is set, at least one byte must be sent.

Implementor's note: data for a t_snd() call with the T_EXPEDITED flag set may not pass data sent previously.

OSI Implementation Specifics

Zero byte TSDUs are not supported. The T_EXPEDITED flag is not a legal flag unless expedited data has been negotiated for this connection.

The t_snd subroutine is not applicable for ConnectionLess Transport Service. See *t_sndudata().*

XX25 Implementation Specifics

The behaviour of the function *t_snd()* remains unchanged. The function can operate in synchronous and asynchronous modes.

In addition to the **T_EXPEDITED** and **T_MORE** flags, the following flags can be set in the argument *flags*:

- **T_X25_D**
  If set in *flags*, the data is sent with the D bit set. This data has to be acknowledged by the peer.
  As with normal *t_snd()* requests, the user may issue multiple *t_snd()* requests with the D–bit set which will be queued by the provider. A separate acknowledgement is generated for each one. The normal flow control mechanism applies: if a *t_snd()* cannot be accepted, the [TFLOW] code is returned.
  Note: As a D–bit send requires end–to–end acknowledgement, it can considerably delay the transmission of further packets.

- **T_X25_Q**
  If set in *flags*, the data is sent as normal qualified data.

- **T_X25_RST**
  If set in *flags*, this indicates to the underlying provider that a request or a confirmation of reset is required.
  The *t_snd()* function returns immediately. If further *t_snd()* calls are accepted while the reset request is being performed they remain pending until the X.25 provider receives the confirmation of reset. This confirmation of reset is not returned to the user. The normal flow control mechanism may result in a subsequent *t_snd()* in synchronous mode blocking, or *t_snd()* call in asynchronous mode returning the [TFLOW] error.
  The cause and diagnostic of a reset request are encoded in the two first octets of the *buf* argument. The cause in the first octet and the diagnostic in the second octet. If the buf argument is NULL or the *buf–>len* is 0, then a cause of 0 and a diagnostic of 0xFA (that means user resynchronisation) are used. If *buf–>len* is 1, the diagnostic is set to 0.

Any cause and diagnostic passed in the *t_snd()* call are ignored by the X.25 provider when sending a reset confirmation.
Data received after a successful *t_snd()* call requesting a reset is data transmitted by the peer after completion of the reset.

- **T_X25_DACK**
  If set in *flags*, this indicates that an explicit acknowledgement of data with the D bit is sent.
- **T_X25_EACK**
  If set in *flags*, this indicates that an explicit acknowledgement of expedited data is sent.
  Note: When either the **T_X25_DACK** or the **T_X25_EACK** flag is set, no other flags can be set, and there must be no user data present on the *t_snd()* call.

An additional error code is also defined:

- [TX25NOTOACK]
  The user attempts to send data acknowledgement but there is currently no pending received data to acknowledge.

See also

t_getinfo(), *t_open*(), *t_rcv*().

t_snddis Subroutine

Purpose

Send user-initiated disconnect request.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
call–>addr.maxlen	/	/
call–>addr.len	/	/
call–>addr.buf	/	/
call–>opt.maxlen	/	/
call–>opt.len	/	/
call–>opt.buf	/	/
call–>udata.maxlen	/	/
call–>udata.len	x	/
call–>udata.buf	?(?)	/
call–>sequence	?	/

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

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

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

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

Valid States

T_DATAXFER, T_OUTCON, T_OUTREL, T_INREL, T_INCON (ocnt >0).

Errors

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

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

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

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

[TBADSEQ] An invalid sequence number was specified, or a null *call* pointer was specified when rejecting a connect request.

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

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

[TLOOK] An asynchronous event has occurred.

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

TCP/IP Implementation Specifics

Since data may not be sent with a disconnect, *call–>udata.len* must be set to zero.

OSI Implementation Specifics

Since at most 64 octets of data may be sent with the disconnect, *call–>udata.len* will have a value of less than or equal to 64.

XX25 Implementation Specifics

The *call–>udata.len* field is

- equal to 0 in basic format,
- in the range 0 to 128 in extended format, format negotiated using the option T_X25_FASTSELECT. (Refer to Appendix C. Bull-enhanced XTI Options.)

The function induces a state transfer to **T_IDLE** and returns at the receipt of the confirmation of the connection release.

In case of PVC-connection mode, *t_snddis()* dissociates the user from the PVC and normally resets the PVC.

See also

t_connect(), *t_getinfo*(), *t_listen*(), *t_open*().

Caveats

t_snddis() is an abortive disconnect. Therefore a *t_snddis*() issued on a connection endpoint may cause data previously sent via t_snd(), or data not yet received, to be lost (even if an error is returned).

t_sndrel Subroutine

Purpose

Initiate an orderly release.

Syntax

#include <xti.h>

int t_sndrel (*fd***)**
int *fd***;**

Description

Parameters	Before call	After call
fd	x	/

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

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

Valid States

T_DATAXFER, T_INREL.

Errors

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

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

[TFLOW] O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting the function at this time.

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

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

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

See also

t_getinfo(), *t_open*(), *t_rcvrel*().

t_sndudata Subroutine

Purpose

Send a data unit.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/
unitdata–>addr.maxlen	/	/
unitdata–>addr.len	x	/
unitdata–>addr.buf	x (x)	/
unitdata–>opt.maxlen	/	/
unitdata–>opt.len	x	/
unitdata–>opt.buf	? (?)	/
unitdata–>udata.maxlen	/	/
unitdata–>udata.len	x	/
unitdata–>udata.buf	x (x)	/

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

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

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

If the *len* field of *udata* is zero, and sending of zero octets is not supported by the underlying transport service, the *t_sndudata*() will return –1 with *t_errno* set to [TBADDATA].

By default, *t_sndudata*() operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NONBLOCK is set (via *t_open*() or *fcntl*()), *t_sndudata*() will execute in asynchronous mode and will fail under such conditions. The process can arrange to be notified of the clearance of a flow control restriction via either *t_look*() or the EM interface.

If the amount of data specified in *udata* exceeds the TSDU size as returned in the *tsdu* field of the *info* argument of *t_open*() or *t_getinfo*(), a [TBADDATA] error will be generated. If *t_sndudata*() is called before the destination user has activated its transport endpoint (see *t_bind*()), the data unit may be discarded.

If it is not possible for the transport provider to immediately detect the conditions that cause the errors [TBADADDR] and [TBADOPT]. These errors will alternatively be returned by *t_rcvuderr*. Therefore, an application must be prepared to receive these errors in both of these ways.

Valid States

T_IDLE.

Errors

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

[TBADDATA] Illegal amount of data. A single send was attempted specifying a TSDU greater than that specified in the *info* argument, or a send of a zero byte TSDU is not supported by the provider.

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

[TFLOW] O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.

[TLOOK] An asynchronous event has occurred on this transport endpoint.

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

[TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

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

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

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

TCP/IP Implementation Specifics

Be aware that the maximum size of a connectionless TSDU varies among implementations.

OSI Implementation Specifics

The *unitdata–>addr* structure specifies the remote TSAP. The ISO connectionless transport service does not support the sending of expedited data. This function is not supported by an ISO–over–TCP transport provider.

For CLTS, TSDU size must be less than NSDU size, since CLTS does not offer segmentation service.

Bull Implementation Specifics

The supported options are listed in Bull-enhanced XTI Option Profiles in Appendix C.)

See also

fcntl(), *t_alloc*(), *t_open*(), *t_rcvudata*(), *t_rcvuderr*().

t_strerror Subroutine

Purpose

Produce an error message string.

Syntax

#include <xti.h>

char *t_strerror (*errnum***)**
int *errnum***;**

Description

Parameters	Before call	After call
errnum	x	/

The *t_strerror*() function maps the error number in *errnum* that corresponds to an XTI error to a language–dependent error message string and returns a pointer to the string. The string pointed to will not be modified by the program, but may be overwritten by a subsequent call to the *t_strerror* function. The string is not terminated by a newline character. The language for error message strings written by *t_strerror*() is implementation–defined. If it is English, the error message string describing the value in *t_errno* is identical to the comments following the *t_errno* codes defined in **<xti.h>**. If an error code is unknown, and the language is English, *t_strerror*() returns the string:

```
"<error>: error unknowm"
```

where <error> is the number supplied as input. In other languages, an equivalent text is provided.

Valid States

ALL – apart from T_UNINIT.

Return Values

The function *t_strerror*() returns a pointer to the generated message string.

See also

t_error().

t_sync Subroutine

Purpose

Synchronise transport library.

Syntax

#include <xti.h>

int t_sync (*fd***)**
int *fd***;**

Description

Parameters	Before call	After call
fd	x	/

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

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

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

If the transport endpoint is undergoing a state transition when *t_sync*() is called, the function will fail.

Valid States

ALL – apart from T_UNINIT.

Errors

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

[TBADF] The specified file descriptor does not refer to a transport endpoint. This error may be returned when the *fd* has been previously closed or an erroneous number may have been passed to the call.

[TSTATECHNG] The transport endpoint is undergoing a state change.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

On successful completion, the state of the transport endpoint is returned. Otherwise, a value of –1 is returned and *t_errno* is set to indicate an error. The state returned is one of the following:

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

See also

dup(), *exec*(), *fork*(), *open*().

t_unbind Subroutine

Purpose

Disable a transport endpoint.

Syntax

#include <xti.h>

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

Description

Parameters	Before call	After call
fd	x	/

The *t_unbind*() function disables the transport endpoint specified by *fd* which was previously bound by *t_bind*(). On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider. An endpoint which is disabled by using *t_unbind*() can be enabled by a subsequent call to *t_bind*().

Valid States

T_IDLE.

Errors

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

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

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

[TLOOK] An asynchronous event has occurred on this transport endpoint.

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

[TPROTO] This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

Return Values

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

See also

t_bind().

Chapter 5. – XTI Name Server Functions

The XTI Name Server facilitates the use of protocol-dependent addresses and options and improves independence of XTI applications from Transport Provider. It is a set of subroutines which help to write XTI applications portable from one Transport Provider to another one.

For each XTI Name Server function, a table is given which summarises the contents of the input and output parameter. The key is given below:

x	The parameter value is meaningful. (Input parameter must be set before the call and output parameter may be read after the call.)
(x)	The content of the object pointed to by the x pointer is meaningful.
?	The parameter value is meaningful but the parameter is optional.
(?)	The content of the object pointed to by the ? pointer is optional.
/	The parameter value is meaningless.
=	The parameter after the call keeps the same value as before the call.
R	Reserved for internal use. The parameter must not be changed by the programmer.

Refer to How to Prepare a Bull-enhanced XTI Application, on page 7-2, to use the appropriate options in compiling and linking the application-program.

List of Bull-enhanced XTI Name Server Functions

Two functions identify a Transport Provider and its associated device (Transport Provider identifier):

- **t_gettp()** Subroutine on page 5-15

  Get the Transport Provider identifier (device name) associated with a specified Transport Provider or a Transport Provider selected automatically.

- **t_getisotp()** Subroutine on page 5-4

  Get the OSI Transport Provider and its identifier according to a Host to be connected to.

Five functions give access to addresses, name (Host and Service) and options independently from the Transport Provider:

- **t_getladdr()** Subroutine on page 5-6

  Get the address of a local XTI Service according to the Transport Provider used.

- **t_getlname()** Subroutine on page 5-8

  Parse a local address according to a Transport Provider and get the XTI Service name which is associated with this address.

- **t_getraddr()** Subroutine on page 5-11

  Get the address to access a given XTI Service on a remote host according to the Transport Provider used.

- **t_getrname()** Subroutine on page 5-13

  Parse a remote address according to a Transport Provider and get the XTI Host and Service name which are associated with this address.

- **t_getopt()** Subroutine on page 5-10

  Initializes an option buffer according to an option profile name.

Another function helps in managing errors.

- **t_error_ns()** Subroutine on page 5-3

  Produce an error message on the standard output.

t_error_ns Subroutine

Purpose

Produce an error message on the error output (stderr).

Syntax

#include <xti_ns.h>

int t_error_ns (*errmsg***)**
char **errmsg***;**

Description

parameters	before call	after call
errmsg	x	/

The **t_error_ns()** function produces on the error output a message which describes the last error encountered during a call to a Name Server function. The argument string *errmsg* is a user-supplied error message that gives context to the error.

The user supplied error message is printed, followed by a colon and a standard error message for the current error defined in **t_errno**.

The **t_errno** number is only set when an error occurs and it is not cleared on successful calls.

Errors

No errors are defined for the **t_error_ns()** function.

Return Values

Upon successful completion, a value of 0 is returned.

Implementation Specifics

This subroutine is part of **xti_api** Software. It is not defined in *X/Open Transport Interface XPG4 CAE Specification Version 2*, but is part of Bull enhancements.

See also

t_error() Subroutine.

t_getisotp Subroutine

Purpose

Get the OSI Transport Provider (COTS) and its identifier according to a Host to be connected to.

Syntax

#include <xti_ns.h>

int t_getisotp (*endsys*,*tp***)**
char * *endsys***;**
struct xtitp **tp***;*

Description

parameters	before call	after call
endsys	x	/
tp–>tp_query	/	/
tp–>tp_id	/	x
tp–>tp_name	/	x
tp–>tp_reserved	R	R

t_getisotp() returns in *tp–>tp_id* an OSI Transport Provider and in *tp–>tp_name* its Transport Provider identifier (device name) according to the Host name defined in *endsys*.

This function allows to define which Transport Provider to use, depending on the name of the Host to be connected to:

- If the Host is found in the /**etc**/**hosts** file, the Transport Provider to use is NetShare (RFC 1006) (*tp–>tp_id= TPID_RFC1006*).
- If the Host is found in the /**etc**/**xtihosts** file, the Transport Provider to use is OSI connection-oriented (*tp–>tp_id= TPID_OSI_COTS*).
- If the Host is found in both files, the NetShare (RFC 1006) Transport Provider is chosen first.

The Transport Provider name is then used by **t_open()** in order to establish a transport endpoint.

Errors

On failure, t_errno is set to the following:

[TENDSYSNOTFOUND] The host name specified by the application specification in endsys is not in the host name database.

Return Value

Upon successfull completion, a value of 0 is returned.
Otherwise, a value –1 is returned and **t_errno** is set to indicate an error.

Implementation Specifics

This subroutine is part of **xti_api** Software. It is not defined in *X/Open Transport Interface XPG4 CAE Specification Version 2*, but is part of Bull enhancements.

t_getisotp() does not apply to an application using TCP/IP, UDP or XX25 Providers.

Files

/etc/hosts

/etc/xtihosts

/etc/xtiprotocols

See also

t_open(), **t_gettp()** Subroutines.

t_getladdr Subroutine

Purpose

Get the address of a local XTI Service according to the Transport Provider used.

Syntax

#include <xti_ns.h>

int t_getladdr (*tp*, *tappl*, *addr***)**
struct xtitp **tp***;*
char **tappl***;*
struct netbuf **addr***;*

Description

parameters	before call	after call
tp –>tp_query	/	/
tp –> tp_id	x	/
tp –> tp_name	/	/
tp –>tp_reserved	R	R
addr –> maxlen	x	/
addr –> len	/	x
addr –> buf	x	(x)
tappl	x	/

t_getladdr() returns in the *netbuf* structure referenced by *addr*, the local address of the XTI Service defined:

tappl the Service Name (null-terminated C–string)
tp–>tp_id the Transport Provider.

- If the Transport Provider specified is OSI (Connection-Oriented or ConnectionLess), NetShare (RFC 1006) or XX25, **t_getladdr()** looks for address in /**etc**/**xtiservices**.
- If the Transport Provider specified is TCP or UDP, **t_getladdr()** looks for the address in /**etc**/**services**.

This local address is then used by **t_bind()** to bind the XTI Service to a transport endpoint.

Errors

On failure, t_errno is set to the following:

[TAPPLNOTFOUND]
The XTI Service specified by *tappl* is not defined in the XTI database.

[TBUFOVFLW] The address to be returned in *addr* is larger than the size specified in the *maxlen* field of the *addr* netbuf structure.

[TNOTSUPPORT]
The *tp* parameter does not refer to a supported Transport Provider.

Return Value

Upon successfull completion, a value of 0 is returned.
Otherwise, a value of –1 is returned and **t_errno** is set to indicate an error.

Implementation Specifics

This subroutine is part of **xti_api** Software. It is not defined in *X/Open Transport Interface XPG4 CAE Specification Version 2*, but is part of Bull enhancements.

Files

/etc/xtiservices

/etc/services

See also

t_bind(), **t_getraddr()** Subroutines.

t_getlname Subroutine

Purpose

Parse a local address according to a Transport Provider and get the XTI Service name which is associated with this address.

Syntax

#include <xti_ns.h>

int t_getlname (*tp*, *addr*, *tappl***)**
struct xtitp **tp***;*
struct netbuf **addr***;*
struct netbuf **tappl***;*

Description

parameters	before call	after call
tp–>tp_query	/	/
tp –> tp_id	x	/
tp –> tp_name	/	/
tp–>tp_reserved	R	R
addr –> maxlen	/	/
addr –> len	x	/
addr –> buf	(x)	/
tappl –> maxlen	x	/
tappl –> len	/	x
tappl –> buf	x	(x)

t_getlname() parses the local address specified in the *netbuf* structure referenced by *addr* according to the Transport Provider specified by *tp–>tp_id*.

If it is correct, **t_getlname()** then looks up in the database the XTI Service name referenced by this address.

- If the Transport Provider specified is OSI (Connection-Oriented or ConnectionLess), NetShare (RFC 1006) or XX25, **t_getlname** looks for address in /**etc**/**xtiservices**.
- If the Transport Provider specified is TCP or UDP, **t_getlname** looks for address in /**etc**/**services**.

The XTI Service name is returned in the *netbuf* structure referenced by *tappl* (*tappl–>len* includes the **NULL** (0) byte which terminates a C–string).

If the XTI Service name is not found in the database, a string of the form 'a.b.c...' is returned, where each element (a,b,c,...) is a hexadecimal representation of one octet of the local address. In this case, –1 will be returned and **t_errno**, will be set to **TAPPLNOTFOUND**.

When no error is returned the string returned in *tappl* may be used by **t_getladdr()**.

Errors

On failure, t_errno is set to the following:

[TBADADDR] The local address in *addr* is not in a recognizable format.

[TAPPLNOTFOUND]
No XTI Service associated with the local address *addr* is present in the database.

[TBUFOVFLW] The Service name to be returned in *tappl* is larger than the size specified in the *maxlen* field of the netbuf structure.

[TNOTSUPPORT]
The *tp* parameter does not refer to a supported Transport Provider.

Return Value

Upon successfull completion, a value of 0 is returned.
Otherwise, a value of –1 is returned and **t_errno** is set to indicate an error.

Implementation Specifics

This subroutine is part of **xti_api** Software. It is not defined in *X/Open Transport Interface XPG4 CAE Specification Version 2*, but is part of Bull enhancements.

Files

/etc/xtiservices

/etc/services

See also

t_getladdr(), **t_getraddr()**, **t_getrname()** Subroutines.

t_getopt Subroutine

Purpose

Initialize an option buffer according to an option profile name and a Transport Provider.

Syntax

#include <xti_ns.h>

int t_getopt (*tp*, *optname*, *optbuf***)**
struct xtitp **tp***;*
char **optname***;*
struct netbuf **optbuf***;*

Description

parameters	before call	after call
tp–>tp_query	/	/
tp–>tp_id	x	/
tp–>tp_name	/	/
tp–>tp_reserved	R	R
optname	x	/
optbuf–>maxlen	x	/
optbuf–>len	/	x
optbuf–>buf	x	(x)

Search an option profile in the XTI database (/**etc**/**xtiopts** file), extract the options available with the specified Transport provider and put them in TLV format in the option buffer *optbuf.*

On return, the option buffer is formatted as described in *X/Open Transport Interface XPG4 CAE Specification Version 2* and can be used as input by:

t_optmgmt() in structure *t_optmgmt,*

t_accept(), **t_connect()**, in structure *t_call,*

t_sndudata in structure *t_unitdata*.

Errors

On failure, t_errno is set to the following:

[TOPTNOTFOUND] The option is not in the XTI Database.

[TBUFOVFLW] The option to be returned in *optbuf* is larger then the size specified in the *optbuf–>maxlen* field.

Return Value

Upon successfull completion, a value of 0 is returned.
Otherwise, a value of –1 is returned and **t_errno** is set to indicate an error.

Implementation Specifics

This subroutine is part of **xti_api** Software. It is not defined in *X/Open Transport Interface XPG4 CAE Specification Version 2*, but is part of Bull enhancements.

Files

/**etc**/**xtiopts**

See also

t_optmgmt(), **t_accept()**, **t_connect()** and **t_sndudata** Subroutines.

t_getraddr Subroutine

Purpose

Get the address to access a given XTI Service on a remote host according to the Transport Provider used.

Syntax

#include <xti_ns.h>

int t_getraddr (*tp*, *endsys*, *tappl*, *addr***)**
struct xtitp **tp***tp***;**
char **endsy***endsys***;**
char **tappl***tappl***;**
struct netbuf **addr***addr***;**

Description

parameters	before call	after call
tp–>tp_query	/	/
tp –> tp_id	x	/
tp –> tp_name	/	/
tp–>tp_reserved	R	/
endsys	x	R
tappl	x	/
addr –> maxlen	x	/
addr –> len	/	x
addr –> buf	x	(x)

t_getraddr() returns in the *netbuf* structure referenced by *addr*, the remote address of the XTI Service defined by:

tappl	the Service Name (null-terminated C–string),
tp–>tp_id	the Transport Provider,
endsys	the Host to access (null-terminated C–string).

- If the Transport Provider specified is OSI (Connection Oriented or ConnectionLess) or XX25, **t_getraddr()** looks for address in the XTI Database (/**etc**/**xtiservices** and /**etc**/**xtihosts** files).
- If the Transport Provider specified is TCP or UDP, **t_getraddr()** looks for the address in the INET Database (/**etc**/**services** and /**etc**/**hosts** files).
- If the Transport Provider specified is NetShare (RFC 1006), the Internet addressing domain for Host declaration is used (/**etc**/**hosts**) and the OSI Transport SELectors for Service definitions (/**etc**/**xtiservices**).

This remote address is then used by **t_connect()** to access a remote XTI Service.

Errors

On failure, t_errno is set to the following:

[TAPPLNOTFOUND]
The XTI Service specified by *tappl* is not defined in the database.

[TENDSYSNOTFOUND]
The Host specified by *endsys* is not in the database.

[TBADNAME] The XTI Service specified by the combination of *tappl* and *endsys* could not be resolved.
[TBADNAME] = **[TAPPLNOTFOUND]** + **[TENDSYSNOTFOUND]**

[TBUFOVFLW] The address to be returned in *addr* is larger than the size specified in the *maxlen* field of the *addr* netbuf structure.

[TNOTSUPPORT]
The *tp* parameter does not refer to a supported Transport Provider.

Return Value

Upon successfull completion, a value of 0 is returned.
Otherwise, a value of –1 is returned and **t_errno** is set to indicate an error.

Implementation Specifics

This subroutine is part of **xti_api** Software. It is not defined in *X/Open Transport Interface XPG4 CAE Specification Version 2*, but is part of Bull enhancements.

Files

/etc/xtihosts

/etc/xtiservices

/etc/hosts

/etc/services

See also

t_connect(), **t_getladdr()** Subroutines.

t_getrname Subroutine

Purpose

Parse a remote address according to a Transport Provider and get the XTI Host and Service name which are associated with this address.

Syntax

#include <xti_ns.h>

int t_getrname (*tp*, *addr*, *endsys*, *tappl***)**
struct xtitp **tp***tp***;**
struct netbuf **tp***addr***;**
struct netbuf **tp***endsys***;**
struct netbuf **tp***tappl***;**

Description

parameters	before call	after call
tp–>tp_query	/	/
tp –> tp_id	x	/
tp –> tp_name	/	/
tp–>tp_reserved	R	R
addr –> maxlen	/	/
addr –> len	x	/
addr –> buf	(x)	/
endsys –> maxlen	x	/
endsys –> len	/	x
endsys –> buf	x	(x)
tappl –> maxlen	x	/
tappl –> len	/	x
tappl –> buf	x	(x)

t_getrname() parses the remote address specified in the *netbuf* structure referenced by *addr* according to the Transport Provider specified by *tp–>tp_id*.

If it is correct, **t_getrname()** then looks up in the database the XTI Host and Service name referenced by this address.

- If the Transport Provider specified is OSI (Connection Oriented or ConnectionLess) or XX25, **t_getrname()** looks for address in the XTI Database (/**etc**/**xtiservices** and /**etc**/**xtihosts** files).
- If the Transport Provider specified is TCP or UDP, **t_getrname()** looks for the address in the INET Database (/**etc**/**services** and /**etc**/**hosts** files).
- If the Transport Provider specified is NetShare (RFC 1006), the Internet addressing domain for Host declaration is used (/**etc**/**hosts**) and the OSI Transport SELectors for Service definitions (/**etc**/**xtiservices**).

If either the Host (*endsys*) and/or the Service name (*tappl*) is not found in the database, a string of the form 'a.b.c...' is returned, where each element (a,b,c,...) is a hexadecimal representation of one octet of the network Host (*endsys*) or Transport Service portions of the address. In this case, –1 will be returned and **t_errno** set to ([**TENDSYSNOTFOUND**], [**TAPPLNOTFOUND**] or [**TBADNAME**]).

When no error is returned the strings returned in *endsys* and *tappl* may be used by **t_getraddr()**.

Errors

On failure, t_errno is set to the following:

[TBADADDR] The remote address in *addr* is not in a recognizable format.

[TBUFOVFLW] The Host name to be returned in *endsys* or the Service name to be returned in *tappl* is larger than the size specified in the *maxlen* field of the *netbuf* structure.

[TAPPLNOTFOUND]
No XTI Service associated with the remote address *addr* is present in the database.

[TENDSYSNOTFOUND]
The Host specified by *endsys* is not in the database.

[TBADNAME] The XTI Service specified by the combination of *tappl* and *endsys* could not be resolved.
[TBADNAME] = **[TAPPLNOTFOUND]** + **[TENDSYSNOTFOUND]**

[TNOTSUPPORT]
The *tp* parameter does not refer to a supported Transport Provider.

Return Value

Upon successfull completion, a value of 0 is returned.
Otherwise, a value of –1 is returned and **t_errno** is set to indicate an error.

Implementation Specifics

This subroutine is part of **xti_api** Software. It is not defined in *X/Open Transport Interface XPG4 CAE Specification Version 2*, but is part of Bull enhancements.

Files

/etc/xtihosts

/etc/xtiservices

/etc/hosts

/etc/services

See also

t_getladdr(), **t_getlname()**, **t_getraddr()** Subroutines.

t_gettp Subroutine

Purpose

Get the Transport Provider identifier (device name) associated with a specified Transport Provider or a Transport Provider selected automatically.

Syntax

#include <xti_ns.h>

int t_gettp (*tp***)**
struct xtitp ******tp***;**

Description

parameters	before call	after call
tp–>tp_query	x	/
tp–>tp_id	x or /	/
tp–>tp_name	/	(x)
tp–>tp_reserved	R	R

t_gettp() returns in *tp–>tp_name* the Transport Provider identifier (device name) associated:

- with the Transport Provider specified in *tp–>tp_id*, if *tp–>tp_query* is equal to 0,
- with a Transport Provider selected automatically in a list if *tp–>tp_query* is not null. *tp–>tp_query* is an OR-combination of
  - **TPID_ANY_COTS** to query the automatic selection of any connection-oriented Transport Provider (**tpid_osi_cots**, **tpid_tcp**, **tpid_rfc1006**, **npid_x25_cons**),
  - **TPID_ANY_COTS_ORD** to query the automatic selection of any connection-oriented with orderly-release Transport Provider (**tpid_tcp**),
  - **TPID_ANY_CLTS** to query the automatic selection of any connection-less Transport Provider (**tpid_osi_clts**, **tpid_udp**),
  - **TPID_ANY** = **TPID_ANY_COTS** + **TPID_ANY_COTS_ORD** +**TPID_ANY_CLTS**
  - **XTINETPATH** to indicate that the automatic selection restarts at the beginning of the list. This flag is reset after execution of the function.

  The list used for the automatic selection is

  - the environment variable **XTINETPATH** if it exists. Example **XTINETPATH=tpid_osi_cots:tpid_tcp:tpid_rfc1006**
  - the XTI Protocols data base /**etc**/**xtiprotocols** if the environment variable **XTINETPATH** does not exist.

The Transport Provider identifier returned by **t_gettp()** is then used by **t_open()** in order to establish a transport endpoint.

Errors

On failure, t_errno is set to the following:

[TNOTSUPPORT]
Two different meanings for specified and automatic selection of a Transport Provider.
The *tp–>tp_id* parameter does not refer to a supported Transport Provider.
There is no Transport Provider defined in the list and corresponding to the type given in *tp–>tp_query*.

Return Value

Upon successfull completion, a value of 0 is returned.
Otherwise, a value of –1 is returned and **t_errno** is set to indicate an error.

Implementation Specifics

This subroutine is part of **xti_api** Software. It is not defined in *X/Open Transport Interface XPG4 CAE Specification Version 2*, but is part of Bull enhancements.

Files

/etc/xtiprotocols

See also

chxti Command.

XTI Environments Configurator on page 3-37.

t_open(), **t_getisotp()** Subroutines.

Chapter 6. – XTI Commands

The XTI commands can be :

- called using SMIT,
- entered manually or used in shell scripts.

Note: The best way to manage the XTI database is to use SMIT, for its look and feel, coherency controls, and help on line. However, when a good knowledge of XTI is acquired and a large number of objects are to be managed, XTI commands may be used directly.

xtihost Command

Purpose

Manages **OSI** and **XX25 Hosts** in the XTI database.

- An **OSI Host** object defines a path within the transport to access a remote host.
- An **XX25 Host** object defines a path within the network to access a remote host.

Both have to be defined only by client-applications.

Syntax

Add a Host entry (OSI or XX25)
xtihost –a [**–p** *ProviderName*] **–h** *HostName* **–n** *NetworkType* [**–ls** *lsap*]
–l *LocalAddress* **–r** *RemoteAddress* [**–u** *Aliases*]

Change a Host entry (OSI or XX25)
xtihost –c [**–p** *ProviderName*] **–h** *HostName* [**–H** *NewHostName*]
[**–n** *NewNetworkType*] [**–ls** *lsap*]
[**–l** *NewLocalAddress*] [**–r** *NewRemoteAddress*]

Remove a Host entry (OSI or XX25)
xtihost –d [**–p** *ProviderName*] **–h** *HostName*

Display all Host entries (OSI or XX25)
xtihost –s [**–p** *ProviderName*]

Description

The **xtihost** administrative command adds, changes, deletes and displays OSI and XX25 Host entries in the XTI database. These Hosts entries are accessed by any XTI (or XX25) client-application running onto OSI (or XX25) and using the XTI Name Server.

Flags

–a Adds a Host (OSI or XX25).
–c Changes a Host (OSI or XX25).
–d Deletes a Host (OSI or XX25).
–s Displays all Hosts (OSI or XX25).

–p *ProviderName*
Specifies the Transport Provider name (**tpid_osi_cots, tpid_osi_clts** or **npid_x25_cons**).
The default value is ”**tpid_osi_cots**”, that is OSI with Connection-Oriented mode of service.

–h *HostName* Specifies the remote Host name.
It is a string of 40 digits maximum.

–H *NewHostName*
Specifies the new Host name (used with **–c** flag).
It is a string of 40 digits maximum.

–n *NetworkType* [or *NewNetworkType*]

For **OSI**, it specifies the network type used to communicate. It may be:
CONS/WAN/PVC (COTS over CONS on Permanent Virtual Circuit)
CONS/WAN/SVC (COTS over CONS on Switched Virtual Circuit)
I_CLNS/LAN (COTS over Inactive CLNS)
CLNS (COTS over CLNS on LAN and WAN – Full OSI conformance)
SPEE (COTS over CONS on WAN or COTS over CLNS on LAN)

For **XX25**, it specifies the circuit type used to communicate. It may be:
PVC (Permanent Virtual Circuit)
SVC (Switched Virtual Circuit)

–ls *lsap*: Significant only on **OSI** Transport for **I_CLNS/LAN** network type, it specifies the Link Service Access Point and may be:
OSI (Full OSI conformance) default value
DSA (Non Full OSI conformance)

–l *LocalAddress* [or *NewLocalAddress*]

Specifies the address through which the connection goes out to the remote host. Depending on the *NetworkType* it is:

For **OSI**,
if **CONS/WAN/PVC,** the name of the **local PVC** to use (8 bytes max)
if **CONS/WAN/SVC,** the **X.121 address** (15 decimal digits max)
if **I_CLNS/LAN,** the **MAC address** (6 bytes max)
if **CLNS**, a **Network Service Access Point** (NSAP, 20 bytes max)
if **SPEE**, a **Network Service Access Point** (NSAP, 20 bytes max)
if **CLNS** or **SPEE**, the local address is optional.

An **NSAP** is built according to the following syntax:
– A string of alpha-numerical digits, enclosed in double quotes, is converted into the ASCII value of the string.
For example ”MYNSAP” will result to the NSAP 0x4d594e534150.
– A string representing hexadecimal digits, is converted into the hexadecimal value of the string.
For example 1234ab will result to the NSAP 0x010203040a0b.
– A string of hexadecimal digits preceded by a ’0x’ remains unchanged.
For example 0x1234ab will result to the NSAP 0x1234ab.

For **XX25**,
if **PVC,** the name of the **local PVC** to use (8 bytes max)
if **SVC,** the **X.121 address** (15 decimal digits max)

–r *RemoteAddress*[or *NewRemoteAddress*]

Specifies the address used to access the remote transport or network. Depending on the *NetworkType* it is:

For **OSI**,
if **CONS/WAN/PVC,** not significant and not to be specified,
if **CONS/WAN/SVC,** the **X.121 address** (15 decimal digits max)
if **I_CLNS/LAN,** the **MAC address** (6 bytes max)
if **CLNS**, a **Network Service Access Point** (NSAP)
if **SPEE**, a **Network Service Access Point** (NSAP)
The syntax of **NSAP** is the same as for *LocalAddress*.

For **XX25**,
if **PVC,** not significant and not to be specified,
if **SVC,** the **X.121 address** (15 decimal digits max)

–u *Aliases* Specifies the alternative names for *HostName*, two aliases maximum separated by a blank.
Each alias is a character string containing no more than 40 characters.

Examples

1. To add a Host named *myxtihost* on an *I_CLNS/LAN* network accessed by the local machine through the SNPA *0x026020000001* and by the remote machine through the SNPA *0x026020000002* via the *OSI* lsap with alternative name *MYXTIHOST*:

```
xtihost –a  –h 'myxtihost'  –l '0x026020000001'
    –r '0x026020000002' –n 'I_CLNS/LAN' –ls 'OSI' –u 'MYXTIHOST'
```

2. To move this Host into an NSAP address *0x800011*, accessed through CLNS network, as *myNEWxtihost*:

```
xtihost –c  –h 'myxtihost'  –H 'myNEWxtihost'  –n 'CLNS'
    –r '0x800011'
```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/etc/xtiprotocols XTI Protocols Data Base,

/etc/xtihosts XTI OSI and XX25 Hosts Data Base.

Prerequisite Information

OSI Addressing in Appendix D.

XX25 Addressing in Appendix E.

Related Information

How to Manage XTI OSI Hosts on page 3-11

How to Manage XX25 OSI Hosts on page 3-19

XTI Name Server library: **t_getraddr()** and **t_getrname()** Subroutines.

xtiserv Command

Purpose

Manages **OSI** and **XX25 Services** in the XTI database.

- An **OSI Service** defines an association between an Application Name (and aliases) and the address (Transport SELector) to be used in order to access this application from the network.
- An **XX25 Service** defines an association between an Application Name (and aliases) and the address (XX25 Subsequent Application Identifier) to be used in order to access this application from the network.

Both must be defined as well by the server which provides it as by the client which uses it.

Syntax

Adds a Service entry (OSI or XX25)

xtiserv –a –v *ServiceName* [**–p** *ProviderName*] **–n** *ServiceAddress* [**– u** *Aliases*]

Changes a Service entry (OSI or XX25)

xtiserv –c –v *ServiceName* [**–p** *ProviderName*] **–V** *NewServiceName*
–P *NewProviderName* [**–n** *NewServiceAddress*]

Removes a Service entry (OSI or XX25)

xtiserv –d –v *ServiceName* [**–p** *ProviderName*]

Displays all Services entries (OSI or XX25)

xtiserv –s [**–p** *ProviderName*] [**–v** *ServiceName*] [**–n** *ServiceAddress*]

Description

The **xtiserv** administrative command adds, changes, deletes and displays OSI and XX25 Services entries in the XTI database. These Service entries are accessed by any XTI (or XX25) application running onto OSI (or XX25) and using the XTI Name Server.

Flags

–a Adds a Service (OSI or XX25).

–c Changes a Service (OSI or XX25).

–d Deletes a Service (OSI or XX25).

–s Displays all Services (OSI or XX25).

–v *ServiceName*
Specifies the Service name.
It is a string of 40 digits maximum.

–V *NewServiceName*
Specifies the new Service name (used with **–c** flag).
It is a string of 40 digits maximum.

–p *ProviderName*

Specifies the Transport Provider name (**tpid_osi_cots, tpid_osi_clts** or **npid_x25_cons**).
The default value is "**tpid_osi_cots**", that is OSI with Connection-Oriented mode of service.

–P *NewProviderName*

Specifies the new Transport Provider name (used with **–c** flag).
The default value is "**tpid_osi_cots**", that is OSI with Connection-Oriented mode of service.

–n *ServiceAddress* [or *NewServiceAddress*]

Specifies the address associated with the Service name, that is:
– for **OSI**, the **OSI Transport SELector** (**TSEL**) associated with the defined Service,
– for **XX25**, the **Subsequent Application Identifier** (**SAI**) associated with the defined Service.
The Service Address, **TSEL** or **SAI**, has a maximum length of 32 bytes and is built according to the following format:
– A string of alpha-numerical digits, enclosed in double quotes, is converted into the ASCII value of the string.
For example "MYTSEL" will result to the Service Address 0x4d595453454c.
– A string representing hexadecimal digits, is converted into the hexadecimal value of the string.
For example 1234ab will result to the Service Address 0x010203040a0b.
– A string of hexadecimal digits preceded by a '0x' remains unchanged.
For example 0x1234ab will result to the Service Address 0x1234ab.
– Specific initialization NULL if the null TSEL or SAI is associated with the Service.

–u *Aliases* Specifies the alternative names for *ServiceName*, two aliases maximum separated by a blank.
Each alias is a character string containing no more than 40 characters.

Example

1. To add a Service named *myxtiappli* with the Transport SELector *0x01ab* onto OSI connection oriented transport Provider: *tpid_osi_cots* with alias *MYXTIAPPLI* enter:

```
xtiserv –a  –v 'myxtiappli'  –n '0x01ab'  –p 'tpid_osi_cots'
      –u 'MYXTIAPPLI'
```

2. To change the Service name of *myxtiappli* to *myNEWxtiappli* and the Transport SELector to *0x040404* enter:

```
xtiserv –c  –v 'myxtiappli'  –p 'tpid_osi_cots'
      –V 'myNEWxtiappli' –n '444'
```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/etc/xtiprotocols XTI Protocols Data Base,

/etc/xtiservices XTI OSI and XX25 Services Data Base.

Prerequisite Information

OSI Addressing in Appendix D.

XX25 Addressing in Appendix E.

Related Information

How to Manage XTI OSI Services on page 3-14

How to Manage XX25 Services on page 3-22

XTI Name Server library:

t_getladdr(), **t_getraddr()**, **t_getlname()**, **t_getrname()** Subroutines.

xtitracelevel Command

Purpose

Manages **XTI Trace Levels**.

Read and modify the XTI trace levels:

- XTI library trace levels in **xtitrace** file,
- XTI library and kernel trace levels in **xticnxtrace** file.

Syntax

xtitracelevel { **–l** | **–k** | **–c** } [**–f** *file*] { **–r** | **–s** (* | *level*) | **–u** (* | *level*) }

Description

The **xtitracelevel** command reads and modifies XTI trace levels, as well as in the user space as in the kernel space.
These trace levels are set on:

- the libraries *xti_api/libxti.a* and *xti_api/libxti_ns.a* if the current toolkit is **XTI_ENHANCED,**
- the libraries *xti_xx25/libxti.a* and *xti_xx25/libxti_ns.a* if the current toolkit is **XX25**.

The trace levels may be defined by default for all the users (administrative trace levels) or specifically for each user.

Flags

–l Select XTI libraries trace levels, that is **10**, **11**, **24**, **27**, **28**, **29** and **30**,

–k Select XTI kernel trace levels, that is **10**, **11**, **24**, **26** and **27**.
To modify these trace levels, the user must have **root** authority.

–c Select XTI libraries and kernel trace levels, that is **10**, **11**, **24**, **26**, **27**, **28**, **29** and **30**.
To modify these trace levels, the user must have **root** authority.

Note: **l**, **k** and **c** are exclusive options, but one of them must be specified.

–f *file* Optional parameter, with default values corresponding to administrative trace levels, that is
/etc/xtitrace using **–l** flag,
/etc/xticnxtrace using **–k** or **–c** flag.
An explicit value for this file, different from */etc/xtitrace* and */etc/xticnxtrace* permits to access to user trace levels. The user has to save this file name in the environment variable **XTI_FILE_TRACE_LEVEL** or **XTI_FILE_TRACE_LEVELCNX**

–r Read trace levels and print them.

–s * | *level* Set all the trace levels (* option) or the trace level *level*.

–u * | *level* Unset all the trace levels (* option) or the trace level *level*.

level may be:
10 States transitions in automatas

11 Entry and return of external XTI lib. func.
Trace of XTI functions, significant only if at least one of these three levels is set:
– **28 CONNECTION functionnalities**

– **29 MANAGEMENT functionnalities**
– **30 DATA TRANSFER functionnalities**

24 Description of I/O parameters values
Significant only if level **11** (Entry and return of external XTI lib. func.) is set. Trace input and output parameters values on the entry and exit of XTI functions,

26 XTI kernel msg to (from) Provider Interface
Trace Provider Interface messages sent by the **xti4mod** streams module to the lower layer and received by **xti4mod** from the lower layer.

27 Data part of messages (limit 4096 bytes)
Trace Data part of messages transmitted through XTI.

28 CONNECTION functionnalities
If level **11** is set, allows to trace the connection functions:
t_accept, t_bind, t_close, t_connect, t_listen, t_open, t_rcvconnect, t_rcvdis, t_rcvrel, t_snddis, t_sndrel, t_unbind.

29 MANAGEMENT functionnalities
If level **11** is set, allows to trace the management functions:
t_alloc, t_error, t_free, t_getinfo, t_getstate, t_look, t_optmgmt, t_sync.

30 DATA TRANSFER functionnalities
If level **11** is set, trace the data transfer functions:
t_rcv, t_rcvudata, t_rcvuderr, t_snd, t_sndudata.

Note: If both **–s** and **–u** options appear in the same command line, setting is done before unsetting.

Example

1. To read administrative XTI libraries and kernel trace levels (description file */etc/xticnxtrace*), enter:

   ```
   xtitracelevel –c –r
   ```

2. To set the administrative libraries trace level 11 and 28 and to unset level 27, enter:

   ```
   xtitracelevel –l –s 11 –s 28 –u 27
   ```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/etc/xtitrace and **/etc/xticnxtrace** XTI Trace Levels Data Base.

Related Information

XTI Trace Configurator, on page 3-28.

xtiopt Command

Purpose

Manages **Option Profiles** in the XTI database.

An **Option Profile** object is a set of XTI options which are relevant to general XTI options or to a specific Transport Provider options. It defines by default options.

Syntax

Add an Option Profile

xtiopt –a –p *ProfileName* [**–o** *OptionLevel*:*OptionName*:*OptionValue[,OptionValue]*]

Change an Option Profile

xtiopt –c –p *ProfileName* [**–P** *NewProfileName*] [**–C –o** *OptionName*
–O *NewOptionLevel*:*NewOptionName*:*NewOptionValue[,NewOptionValue]*]

Delete an Option in a Profile

xtiopt –c –p *ProfileName* **–D –o** *OptionName*

Add an Option in a Profile

xtiopt –c –p *ProfileName* **–A –o** *OptionLevel*:*OptionName*:*OptionValue[,OptionValue]*

Delete an Option Profile

xtiopt –d –p *ProfileName*

Display all Option Profiles

xtiopt –s [**–p** *ProfileName*]

Description

The **xtiopt** administrative command adds, changes, deletes and displays Option Profiles entries in the XTI database. These Option Profiles entries are accessed by any XTI application using the XTI Name Server.

Flags

–a	Adds a Profile
–c	Changes a Profile
–d	Deletes a Profile
–s	Displays all Profiles

–p *ProfileName* Specifies the Profile name.
It is a string of 20 digits maximum.

–P *NewProfileName*
Specifies the new Profile name (used with **–c** flag).
It is a string of 20 digits maximum.

–A Adds an Option into a specified Profile (used with **–c** flag)

–D Deletes an Option from a specified Profile (used with **–c** flag)

–C Changes an Option in a specified Profile (used with **–c** flag)

–o *OptionLevel*:*OptionName*:*OptionValue[,OptionValue]*
Specifies an Option entity

–O *NewOptionLevel*:*NewOptionName*:*NewOptionValue[,NewOptionValue]*
Specifies the new Option (used with **–c** flag)

Refer to Options in Appendix C. for a complete list of the XTI Options which can be used.

Example

To add a Profile named Test1 with three XTI_GENERIC Options enter:

```
xtiopt –a –p Test1 –o XTI_GENERIC:XTI_DEBUG:NULL
xtiopt –c –p Test1 –A –o XTI_GENERIC:XTI_LINGER:T_YES,12
xtiopt –c –p Test1 –A –o XTI_GENERIC:XTI_SNDBUF:12
```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/**etc**/**xtiopts** XTI Option Profiles Data Base.

Related Information

XTI Option Profile Configurator, on page 3-25.

XTI Name Server library: **t_getopt()** Subroutine.

chxti Command

Purpose

Changes the current XTI attributes.

Syntax

chxti *Attribute=Value*

Description

The **chxti** command changes the current XTI attributes:

adtfile is the application development toolkit configuration file. The **/etc/xlC.cfg** is used by default, but any C compiler resource configuration file may be used instead.

adtname is the application development toolkit name. Three possible values:

- **XTI_BASE**, the AIX-issued XTI library, embedded with the Operating System.
- **XTI_ENHANCED**, the Bull-enhanced XTI which includes:
  - access to OSI Connection–Oriented Transport, OSI ConnectionLess Transport, NetShare (RFC 1006), TCP, UDP,
  - Name Server facilities for Hosts, Services and Option Profiles,
  - Traces facilities,
  - Troubleshooting tools.
- **XX25**, which includes **XTI_ENHANCED** and the **XX25** library to access directly the X.25 network.

netpath is the list of Transport Providers used by the primitive **t_gettp()** for the automatic selection of a Transport Provider. It is a list of Transport Provider names (defined in **/etc/xtiprotocols**) separated by colon.
This list is saved in the environment variable **XTINETPATH.**

Note: Any user (other than root) can use **chxti** to define its own environment. In this case, the specific value '/' can be used to inherit directly the attributes already defined by root.

Flags

None

Example

To switch to the Bull-enhanced XTI libraries:

```
chxti adtname=XTI_ENHANCED
```

To be able to run XTI name service application first on OSI then on TCP:

```
chxti 'netpath=tpid_osi_cots:tpid_tcp'
```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/etc/xlC.cfg Default Common C compiler configuration file

/etc/xtiprotocols

Related Information

XTI Environments Configurator on page 3-37.

lsxti Command.

t_gettp Function.

lsxti Command

Purpose

Displays the current XTI attributes.

Syntax

lsxti

Description

The **lsxti** command lists the current XTI attributes:

adtfile is the application development toolkit configuration file.

adtname is the application development toolkit name:
XTI_BASED, **XTI_ENHANCED** or **XX25**.

netpath is the list of Transport Providers used by the primitive **t_gettp()** for the automatic selection of a Transport Provider.

Flags

None

Example

```
lsxti

adtfile : Name of C compiler resource file:  /etc/xlC.cfg
adtname : Name of XTI developement toolkit:  XTI_ENHANCED
netpath : Transport provider path         :  NULL
```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/etc/xlC.cfg Default Common C compiler configuration file

/etc/xtiprotocols

Related Information

XTI Environments Configurator on page 3-37

chxti Command.

t_gettp Function.

Chapter 7. – Cookbook

Note: The examples described in this cookbook are provided with Bull-enhanced XTI and may be run after the local Host has been configured using the XTI configurator.

How to Prepare a Bull-enhanced XTI Application

One of these two application development toolkits has to be chosen using the SMIT configurator or the **chxti** command:

1. **XTI_ENHANCED**, to develop XTI applications onto TCP/IP, OSI Stack or NetShare (RFC 1006),
2. **XX25**, to develop XX25 applications as well as XTI applications onto TCP/IP, OSI Stack or NetShare (RFC 1006).

Note: The **XX25** toolkit allows to develop applications portable on any of the communications providers: X.25, TCP/IP, OSI Stack and NetShare (RFC 1006). However, if the application is designed to run only on TCP/IP, OSI Stack and NetShare (RFC 1006), it is better to use the **XTI_ENHANCED** toolkit.

Using the XTI_ENHANCED Toolkit

List of Components

Libraries

Two shared libraries are provided:

/usr/lib/xti_api/libxti.a	for the XTI functions
/usr/lib/xti_api/libxti_ns.a	for the XTI Name Server functions

Two thread-safe libraries are provided:

/usr/lib/xti_api/libxti_r.a	for the XTI functions
/usr/lib/xti_api/libxti_ns_r.a	for the XTI Name Server functions

Include Files

/usr/include/xti_api/xti.h	for definition of the structures and constants used by the XTI functions
/usr/include/xti_api/xti_ns.h	for definition of the structures and constants used by the XTI Name Server functions.

Examples

Program examples (**clts**, **cots**, **cots_r**, **poll** and **select**) are provided in the directory **/usr/lpp/xti_api/examples/binop_c**

Compilation and Link Options

The simplest way to compile and link an XTI application (`file.c`) using the Bull-enhanced XTI library and XTI Name Server library is to use the standard C Compiler resource file **/etc/xlC.cfg** and run the following commands:

- to use the shared libraries:
  ```
  cc file.c -F:xticc
  ```
- to use the thread-safe libraries:
  ```
  cc file.c -F:xticc_r
  ```
- to use the non-shared libraries:
  ```
  cc file.c -F:xticc -bnso
  ```

Using the XX25 Toolkit

List of Components

Libraries

Two shared libraries are provided:

/usr/lib/xti_xx25/libxti.a	for the XTI/XX25 functions
/usr/lib/xti_xx25/libxti_ns.a	for the XTI Name Server functions

Two thread-safe libraries are provided:

/usr/lib/xti_xx25/libxti_r.a	for the XTI/XX25 functions
/usr/lib/xti_xx25/libxti_ns_r.a	for the XTI Name Server functions

Include Files

/usr/include/xti_xx25/xti.h	for definition of the structures and constants used by the XTI/XX25 functions
/usr/include/xti_xx25/xx25addr.h	for definition of the structures and constants used by the XX25 addressing.
/usr/include/xti_xx25/xti_ns.h	for definition of the structures and constants used by the XTI Name Server functions.

Examples

Program examples (**cons**) are provided in the directory **/usr/lpp/xti_api/examples/binop_c**

Compilation and Link Options

The simplest way to compile and link an XTI/XX25 application (`file.c`) using the XX25 library and XTI Name Server library is to use the standard C Compiler resource file **/etc/xlC.cfg** and run the following commands:

- to use the shared libraries:
  ```
  cc file.c -F:xx25
  ```
- to use the thread-safe libraries:
  ```
  cc file.c -F:xx25_r
  ```
- to use the non-shared libraries:
  ```
  cc file.c -F:xx25 -bnso
  ```

How to Manage XTI Options

XTI Options are parameters defining the conditions of communications between two XTI applications or transport users. All options have default values, but these values can be negotiated by a transport user. An **XTI Option** is specified by:

- a *level*,
  - **XTI_GENERIC** for options negotiated between XTI and Transport,
  - **ISO_TP** for options on OSI and NetShare (RFC 1006),
  - **INET_TCP**, **INET_UDP**, or **INET_IP** for options on TCP/IP and UDP/IP,
  - **X25_NP** for options on XX25,
- a *name*, which identifies the option within the level,
- a *value*.

The XTI Options are negotiated using the following functions:

- *t_optmgmt* ( ), *t_accept* ( ), *t_connect* ( ) and *t_connect* ( ),
- *t_listen* ( ), *t_*rcvconnect ( ), *t_rcvudata* ( ) and *t_rcvuderr* ( ),

Refer to **Appendix C. Options** for a complete list of the XTI Options available and to *X/Open Transport Interface XPG4 CAE Specification Version 2* **Chapter 5. The Use of Options**.

An **Option Profile** is a set of XTI options, which is configured using:

- **XTI Option Profile Configurator**, on page 3-25,
- or directly the **xtiopt** command, on page 6-10,

It defines default values.

The **t_getopt ( )** Name Server function initializes an option buffer according to an Option Profile name and a Transport Provider (only the options relative to this Tranport Provider are put in the option buffer). The option buffer may then be used as input parameter for the functions *t_optmgmt* ( ), *t_accept* ( ), *t_connect* ( ) and *t_connect* ( ).

Option Profile allows to develop applications using parameters which are open to customization by the end-user:

- the application uses specified Option Profiles which are configurated by the user according to his needs,
- the application may be multiprotocol-developped, the end-user defining the Transport Provider to use.

How to Use XTI Traces

XTI Traces may be configured and used through XTI Trace Configurator.

How to Configure XTI Trace Levels

1. There is a default configuration of XTI Trace levels: **Warning and protocol errors**

2. Global XTI Trace levels may be configured by the administrator, they act as default trace levels for any user.

   Run the XTI Configurator using the command:

   #smit xtitrace

   Select the entry:

   Change/Show Administrative Trace Levels

   then one of the entry

   Change/Show XTI Libraries Trace Levels
   Change/Show XTI Libraries and Kernel Trace Levels
   Change/Show XTI Kernel Trace Levels

   according to events and data to be traced (user-space and/or kernel-space)

3. User XTI Trace levels may be configured by any user, they act as default trace level for any process run by this user.

   Run the XTI Configurator using the command:

   #smit xtitrace

   Select the entry:

   Change/Show User Trace Levels

   then one of the entry

   Change/Show XTI Libraries Trace Levels
   Change/Show XTI Libraries and Kernel Trace Levels

   according to events and data to be traced (user-space or kernel-space)

Note: These user trace levels are saved in user files whose full path name are defined in shell environment variables **XTI_FILE_TRACE_LEVEL** and **XTI_FILE_TRACE_LEVELCNX**.
If these Shell environment variables **XTI_FILE_TRACE_LEVEL** and **XTI_FILE_TRACE_LEVELCNX** do not exist, the default trace levels defined by the administrator are used.

4. Application-specific XTI Trace levels may be configured directly in a program using the **t_optmgmt()** function, described on page 4-39. This is useful to trace a specific connection after an event has been detected.

Warning: Any user can set his own trace levels, but when recording traces, user levels **and** super-user levels are both taken into account.

Warning: When the application needs no longer to be traced, all the trace levels (user and super-user) must be disabled in order to improve performances.

How to Run XTI Traces

5. **Start recording trace events**

   To record only the XTI events, run the XTI Configurator using the command:

   #smit xtitraceuse

   Select the entry:

   Start XTI Trace

   XTI events may be recorded using **Common Trace Utility** and indicating the XTI hook_id: 906.

6. **Run the XTI application**

7. **Stop recording trace events**

   Run the XTI Configurator using the command:

   #smit xtitraceuse

   Select the entry:

   Stop XTI Trace

8. **Display of the trace report**

   Run the XTI Configurator using the command:

   #smit xtitraceuse

   Select the entry:

   Generate an XTI Trace Report

Example of XTI traces

1. Configuration of XTI Trace levels

   Run the XTI Configurator using the command:

   #smit xtitrace
   Change/Show Administrative Trace Levels
   Change/Show XTI Libraries Trace Levels

   Select the following trace levels

   Warning and protocol errors: yes
   CONNECTION functionnalities: yes
   MANAGEMENT functionnalities: yes
   DATA TRANSFER functionnalities: yes
   Entry and return of external XTI lib. func.: yes
   Description of I/O parameters values: yes
   States transitions in automatas: no
   Data part of messages (limit 4096 bytes): yes

2. Start recording trace events

   #smit xtitraceuse

   Start XTI Trace

3. Run the command to be traced

   #bench –x –o –hpean_x25

   The server host name is such defined in /**etc**/**xtihosts**:

```
Remote Host Address  Local Host Address  Lsap   Netser  Host name
19119033344           19119033344         0xfe   0x0101  pean_x25
```

4. Stop recording trace events

   #smit xtitraceuse
   Stop XTI Trace

5. Display of the trace report

   #smit xtitraceuse
   Generate an XTI Trace Report

   Trace report for the *t_connect* ( ) function :

```
XTI-API Traces [764 usec]  LIBXTI  pid=6674  [3]  ==>  t_connect  fd=3
sndcall=0x20004E28  rcvcall=0x20005708
    sndcall->addr.maxlen=120
    sndcall->addr.len=64
    sndcall->addr.buf=
0000  00 00 00 01 00 00 00 04 01 02 03 04 00 00 00 04  ................
0010  00 00 00 04 00 00 00 01 00 00 00 02 00 00 00 14  ................
0020  FF 01 0B xx xx xx xx xx x0 00 00 0B yy yy yy yy  ......34@......3
0030  yy y0 00 00 00 00 00 03 00 00 00 01 FE 00 00 00  4@..............
    sndcall->opt.maxlen=2000
    sndcall->opt.len=0
    sndcall->opt.buf=0x20004ED8
    sndcall->udata.maxlen=64
    sndcall->udata.len=0
    sndcall->udata.buf=0x200056B8
    sndcall->sequence=0
    rcvcall->addr.maxlen=120
    rcvcall->addr.len=0
    rcvcall->addr.buf=0x20005738
    rcvcall->opt.maxlen=2000
    rcvcall->opt.len=0
    rcvcall->opt.buf=0x200057B8
    rcvcall->udata.maxlen=64
    rcvcall->udata.len=0
    rcvcall->udata.buf=0x20005F98
    rcvcall->sequence=0
```

The dump of `sndcall->addr.buf` can be decoded as follows :

```
00 00 00 01   : TTSEL type
00 00 00 04   : TSEL length
01 02 03 04   : TSEL value
00 00 00 04   : TNETSRV type
00 00 00 04   : TNETSRV length
00 00 00 01   : TNULLCLNP
00 00 00 02   : TNSAP type
00 00 00 14   : TNSAP length
FF 01         : WAN_SVC
0B xx xx xx xx xx x0 00 00  :
                length (in half byte)  + value of calling X121 address
                                        (+ padding to 0)
0B yy yy yy yy yy y0 00 00  :
                length (in half byte)  + value of called X121 address
                                       (+ padding to 0)
00 00 00 03   : TLSAP type
00 00 00 01   : TLSAP length
FE 00 00 00   : TLSAP value
```

Overview of an XTI Connection-oriented Mode Service

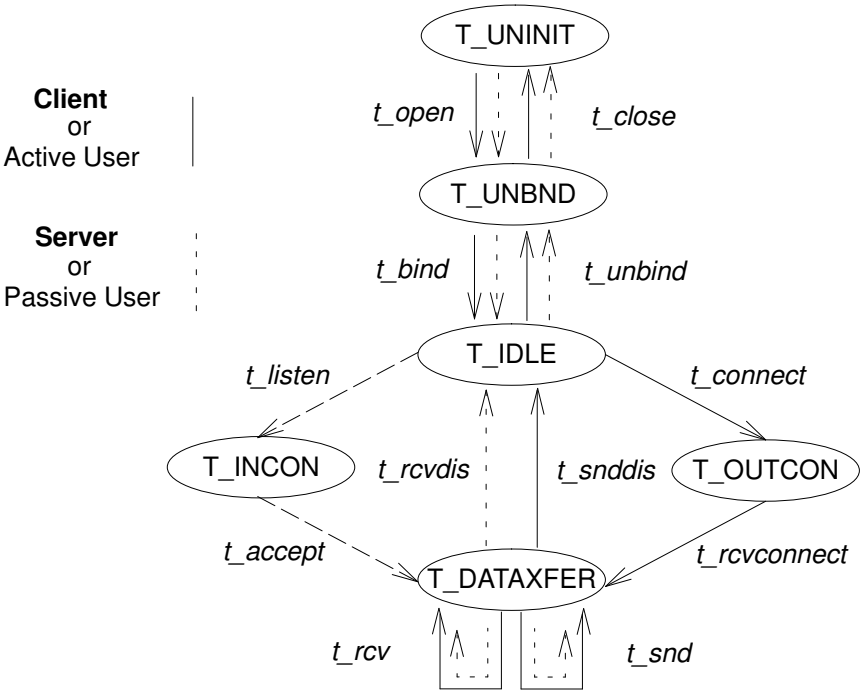
Connection-oriented Mode is circuit–oriented and enables data to be transmitted over an established connection in a reliable, sequenced manner. This service:

- enables the negotiation of the parameters and options that govern the transfer of data,
- provides an identification mechanism that avoids the overhead of address resolution and transmission during the data transfer phase,
- provides a context in which successive units of data, transferred between peer users, are logically related.

This Connection-oriented Mode service is attractive for applications which require relatively long–lived, datastream–oriented interactions.

In a Connection-oriented Mode service, the connection is based on a client–server relationship between two transport users.

The example of Connection-oriented Mode service described in this state machine is not meant to show all the functions that may be called, but rather to hightlight the main functions that request a particular service request.



	Client	Server
Local Management : Initialisation	t_open() t_bind()	t_open() t_bind()
Connection Establishment	t_connect() t_rcvconnect()	t_listen() t_accept()
Data Transfer	t_snd() t_rcv()	t_rcv() t_snd()
Connection Release	t_snddis()	t_rcvdis()
Local Management : De–initialisation	t_unbind() t_close()	t_unbind() t_close()

Figure 5. Sequence of XTI functions in Connection-oriented Mode

A Connection-oriented Mode transport service, consists of four phases of communication described step by step for the Client and for the Server:

- Local Management, on page 7-10, to establish a communication channel with the Transport Provider and establish its identity or address,
- Connection Establishment, on page 7-17, to open a logical connection, or virtual circuit, with the remote user,
- Data Transfer, on page 7-26, to exchange data over the connection,
- Connection Release, on page 7-31, to end the connection.

The example is described using Name Server according to this scenario:

1. The Client issues a connection request to the Server.
2. The Server accepts the connection and sends a file to the Client.
3. The Client requests for disconnection.

By default, the connection is a loopback at Internet level (i.e. the connection request is not sent over the network) because the called NSAP is a local NSAP (value 02).

This local NSAP is mapped by the entry *localhost* in /**etc**/**xtihosts** file.
The TSEL used, '0x01020301', is mapped by the entry *cots_server* in /**etc**/**xtiservices**.
Note that the NSAP (02) must be configured in the OSI stack.

The directory /**usr**/**lpp**/**xti_api**/**examples**/**binop_c**/**cots** contains the source files of the complete programs:

cots_client.c Client-program of an XTI Connection-oriented Mode service using Name Server

cots_server.c Server-program of an XTI Connection-oriented Mode service using Name Server

cots_cnons.c Client-program of an XTI Connection-oriented Mode service without using Name Server

cots_snons.c Server-program of an XTI Connection-oriented Mode service without using Name Server

Local Management in an XTI Connection-oriented Mode Service

The local management phase defines local operations between a transport user and a transport provider. Here are described the operations relative to initialization of a Connection-oriented Mode service.

Before establishing a connection, any transport user (client or server) must establish a communication channel with the transport provider. Each channel between a transport user and transport provider is a unique endpoint of communication, and is called the transport endpoint. The **t_open()** routine enables a user to choose a particular transport provider which supplies the Connection-oriented Mode services, and establishes a channel with the transport endpoint, as shown in the figure.

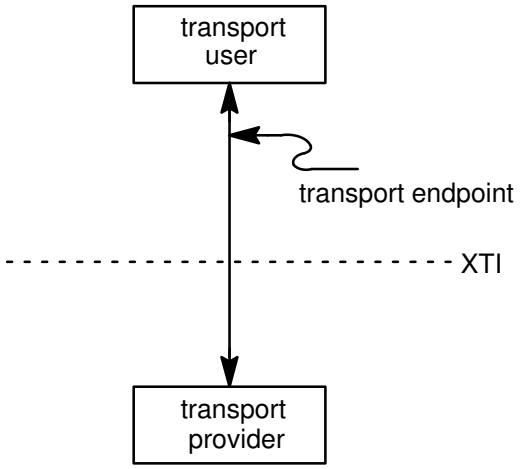


Figure 6. Channel Between Transport User and Transport Provider

t_open() returns the default characteristics of the transport provider associated with the transport endpoint:

addr	maximum size of a transport address
options	maximum bytes of protocol–specific options that may be passed between the transport user and transport provider
tsdu	maximum message size that may be transmitted
etsdu	maximum expedited data message size that may be sent over a transport connection
connect	maximum number of bytes of user data that may be passed between users during connection establishment
discon	maximum bytes of user data that may be passed between users during the abortive release of a connection
servtype	the type of service supported by the transport provider, in this case **T_COTS** (Connection-Oriented Transport Service without orderly release facility) or **T_COTS_ORD** (Connection-Oriented Transport Service with orderly release facility)
flags	other info about the Transport Provider

Note: The characteristics associated with negotiated options may change after a transport endpoint has been opened (Option negotiation is described in Connection Establishment, on page 7-17).
t_getinfo() can be called at any moment to retrieve the current characteristics of a transport endpoint.

Then the user has to communicate its identity to the transport provider, using **t_bind()** routine.
t_bind() binds a transport address to the transport endpoint. In addition, for servers, it informs the transport provider that the transport endpoint will be used to listen for incoming connect requests, also called connect indications.

Note: Each transport provider uses its own mechanism for identifying users and defining transport address.

During the local management phase, **t_optmgmt()** may be used to negotiate the values of protocol options with the transport provider. Each transport provider defines its own set of negotiable protocol options, which may include such information as Quality–of–Service parameters. Because of the protocol–specific nature of options, only applications written for a particular protocol environment are expected to use this facility.

The following table summarizes the XTI routines which support local operations.

Functions	Description
t_alloc()	Allocates XTI data structures
t_bind()	Binds a transport address to a transport endpoint
t_close()	Closes a transport endpoint
t_error()	Produces an XTI error message
t_free()	Frees structures allocated by t_alloc
t_getinfo()	Returns a set of parameters associated with a particular transport provider
t_getprotaddr()	Returns the local and remote protocol addresses associated with the transport endpoint
t_getstate()	Returns the state of a transport endpoint
t_look()	Returns the current event on a transport endpoint
t_open()	Establishes a transport endpoint connected to a chosen transport provider
t_optmgmt()	Negotiates protocol–specific options with the transport provider
t_strerror()	Produces an XTI error message string
t_sync()	Synchronizes a transport endpoint with the transport provider
t_unbind()	Unbinds a transport address from a transport endpoint

Example of a Client-Program for Local Management in an XTI Connection-oriented Mode Service.

```
/*---------------------------------*/
/* -- LMGMT : LOCAL MANAGEMENT -- */
/*---------------------------------*/

if (iso == TRUE) {
        tp.tp_id = TPID_OSI_COTS;
}
else {
        if (rfc == TRUE) {
                tp.tp_id = TPID_RFC1006;
        }
        else {
                tp.tp_id = TPID_TCP;
        }
}
if (t_gettp(&tp) < 0) {
        t_error_ns("t_gettp failed");
        exit(3);
}
if ((fd = t_open(tp.tp_name, O_RDWR, &info)) < 0) {
        t_error("t_open failed for fd");
        exit(4);
}
if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed for fd");
        exit(5);
}
```

In this example, the user can choose between three connection-oriented transport protocols: TCP/IP, NetShare (RFC 1006) or OSI connection transport protocol classes 0,2,3,4.

The first argument of **t_open()** is the path name of a file system node that identifies the transport provider. The corresponding path name is obtained by a call to the **t_gettp()** function.

t_gettp() is a function of the XTI Name Server, added to the standard XTI library to facilitate manipulation of protocol addresses. Refer to XTI Name Server Functions, on page 5-2, to have more details on these functions and the use of the associated **xti_ns** library.

In the *tp_name* field of its **xtitp** structure, **t_gettp()** returns the path name corresponding to a transport protocol code chosen between the various transport protocol codes defined in the include file **<xti_ns.h>**:

- The path name corresponding to TCP/IP is: ***/dev/xti/tcp***.
- The path name corresponding to OSI is: ***/dev/xti/cotp***.
- The path name corresponding to NetShare (RFC 1006) is: ***/dev/xti/tp1006***.

The **t_info** structure, used as third parameter of **t_open()** function, returns the service characteristics of the transport provider to the user. This information is necessary to write protocol independent software. Basic rules for using this information are presented in *X/Open Transport Interface XPG4 CAE Specification Version 2*, Appendix C.

- TCP/IP supports a T_COTS_ORD service (Connection Oriented transport Service with orderly release).
- OSI supports a T_COTS service (Connection Oriented transport Service without orderly release).

For simplicity the client and server in this example do not exchange user data during either connection establishment or abortive release.

The return value of **t_open()** is an identifier for the transport endpoint which is used by all subsequent XTI function calls.

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

The address associated with a server transport endpoint is important, because it is the address used by all clients to access the server. Whereas the typical client does not care what its own address is, because no other process should try to access it. That is the case in this example, where the second and third parameters of **t_bind()** are set to NULL. A NULL second parameter directs the transport provider to choose an address for the user. A NULL third parameter indicates that the user does not care what address was assigned to the endpoint. Furthermore, from XPG4 a successful call to **t_bind()** with the second parameter returned not NULL implies that the endpoint has been bound to the requested address.

If either **t_open()** or **t_bind()** fail, the program calls **t_error()** to print an appropriate error message to **stderr**. If any XTI routine fails, the global integer **t_errno** is assigned to an appropriate transport error value. A set of such error values has been defined (in **<xti.h>**) for the X/OPEN Transport Interface, and **t_error()** prints an error message corresponding to the value in **t_errno**. This routine is analogous to **perror()**, which prints an error message based on the value of **errno**. If the error associated with a transport function is a system error, **t_errno** is set to **TSYSERR**, and **errno** is set to the appropriate value.

Example of a Server-Program for Local Management in an XTI Connection-oriented Mode Service.

```
        /*--------------------------------*/
        /* -- LMGMT : LOCAL MANAGEMENT -- */
        /*--------------------------------*/

        if (iso == TRUE) {
                tp.tp_id = TPID_OSI_COTS;
        }
        else {
                if (rfc == TRUE) {
                        tp.tp_id = TPID_RFC1006;
                }
                else {
                        tp.tp_id = TPID_TCP;
                }
        }
        if (t_gettp(&tp) < 0) {
                t_error_ns("t_gettp failed");
                exit(3);
        }
        if ((listen_fd = t_open(tp.tp_name, O_RDWR, &info)) < 0) {
                t_error("t_open failed for listen_fd");
                exit(4);
        }
        if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND, T_ALL))
                  == NULL) {
                t_error("t_alloc failed for bind");
                exit(5);
        }
        if (t_getladdr(&tp, servername, &bind->addr) < 0) {
                t_error_ns("t_getladdr failed");
                exit(6);
        }

        /* the server endpoint is used to listen for connect indication */
        bind->qlen = 1;
        if ((bindret = (struct t_bind *)t_alloc(listen_fd, T_BIND, T_ALL))
                     == NULL) {
                t_error("t_alloc failed for bindret");
                exit(7);
        }
        if (t_bind(listen_fd, bind, bindret) < 0) {
                t_error("t_bind failed for listen_fd");
                exit(8);
        }
#ifndef XTI4
        /* was the correct address bound ? */
        if ((bindret->addr.len != bind->addr.len) ||
             (memcmp(bindret->addr.buf, bind->addr.buf, bind->addr.len))) {
                fprintf(stderr, "t_bind bound wrong address\n");
                exit(9);
        }
#endif
        if (trace) {
                fprintf(stdout,"server : ");
                for (i=0, pt=bindret->addr.buf; i<bindret->addr.len; i++, pt++)
                        fprintf(stdout, "0x%x", *pt);
                fprintf(stdout,"\n");
        }
```

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

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

where *:*

- the *addr* field describes the address to be bound; it is specified using a **netbuf** structure that contains the following members:

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

  – *maxlen* indicates the maximum bytes the buffer can hold (and need only to be set when data is returned to the user by an X/OPEN Transport Interface routine)

  – *len* specifies the bytes of data in the buffer,

  – *buf* points to a buffer containing the data which identifies a transport address.

  The structure of addresses is expected to vary among each protocol implementation under the X/OPEN Transport Interface, but the **netbuf** structure is intended to support any such structure.

- the *qlen* field indicates the maximum outstanding connect indications that may arrive at this endpoint

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

t_alloc() is called to allocate the **t_bind** structure needed by **t_bind()** function call. **t_alloc()** takes three parameters:

- The first parameter (**integer**) is a file descriptor which references a transport endpoint. It is used to access the characteristics of the transport provider.
- The second parameter (**integer**) identifies the appropriate XTI structure to be allocated.
- The third parameter (**integer**) specifies which **netbuf** buffers, if any, should be allocated for that structure.

In this example, T_ALL given as third parameter specifies that all **netbuf** buffers associated with the structure should be allocated, and causes the *addr* buffer to be allocated. The size of this buffer is determined from the transport provider characteristic that defines the maximum address size. The *maxlen* field of this **netbuf** structure is set to the size of the newly allocated buffer by **t_alloc()**.

t_getladdr() is a function of the XTI Name Server, added to the standard XTI library to facilitate manipulation of protocol addresses. Refer to XTI Name Server Functions, on page 5-2, to have more details on these functions and the use of the associated **xti_ns** library.

Given the specific transport provider as first parameter and a string identifying the current application name as second parameter **t_getladdr()** returns the corresponding well structured protocol address in a **netbuf** structure which can be used in a call to **t_bind()**.

- For TCP/IP, the *buf* field of the **t_bind** *addr* field must correspond to a **sockaddr_in** structure as defined for the socket interface in the **<netinet/in.h>** include file. The *len* field must be set accordingly, i.e. sizeof (struct **sockaddr_in**).
- For OSI, the *buf* field of a **t_bind** *addr* field must correspond to an OSI transport selector, i.e. an identifier of 31 bytes maximum.

The server in this example processes connect indications one at a time, so *qlen* is set to 1. The address information is then assigned to the first newly allocated **t_bind** structure. This **t_bind** structure is used to pass information to **t_bind()** in the second parameter. The second allocated **t_bind** structure is used to return information to the user in the third parameter.

On return, the **t_bind** structure contains the address which was bound to the transport endpoint. If the provider cannot bind the requested address (perhaps because it had been bound to another transport endpoint), it will choose another appropriate address.

Note: Each transport provider manages its address space differently. Some transport providers allows a single transport address to be bound to several transport endpoints, while others requires a unique address per endpoint. XTI supports either choice. Based on its address management rules, a provider determines if it can bind the requested address. If it cannot, the TADDRBUSY error is returned

If **t_bind()** succeeds, the provider begins queueing connect indications. The phase of connection establishment is initiated.

Connection Establishment in an XTI Connection-oriented Mode Service

The connection establishment phase enables two users to create a connection, or virtual circuit, between each other as demonstrated in this illustration.

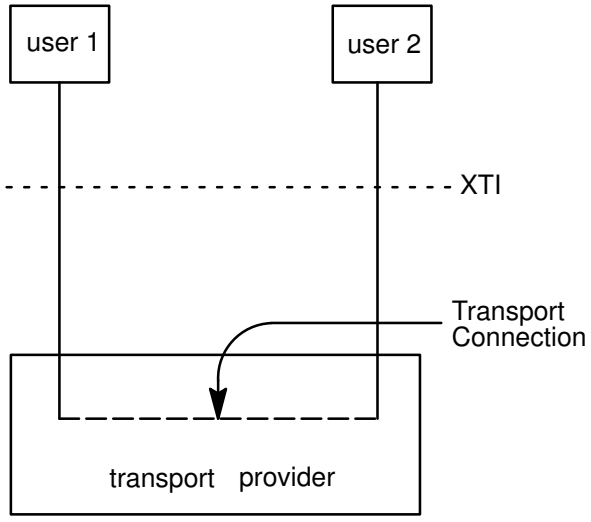


Figure 7. Transport Connection

The connection establishment procedures highlight the distinction between clients and servers:

- The client initiates the connection establishment procedure by requesting a connection to a particular server using **t_connect**().
- The server is notified of the client request by calling **t_listen**(). The server may either accept or reject the client request. It calls **t_accept**() to establish the connection or calls **t_snddis**() to reject the request.
- The client is notified of the server decision:
  - with OSI, when **t_connect()** completes,
  - with TCP, **t_connect()** completion is not a connect indication because it is generated directly by the TCP provider itself. Only the rejection of the connection request by the server through **t_snddis()** is significant.

Note: A **t_snddis()** called to reject a connect indication is analogous to a **t_snddis()** on an established connection.

The following table summarizes the XTI routines which support connection establishment.

Functions	Description
t_accept	Accepts a connection request from a transport user.
t_connect	Requests a connection with another transport user.
t_snddis	Rejects a connection request.
t_listen	Retrieves an indication of a connect request from a transport user.
t_rcvconnect	Completes connection establishment if a **t_connect()** was called in asynchronous mode (see note below).

XTI offers two facilities during connection establishment which may not be supported by all transport providers:

1. the transfer of data between client and server when establishing the connection. The client can send data to the server when it requests a connection. This data is passed to the server by **t_listen()**. Similarly, the server can send data to the client when it accepts or rejects the connection. The *connect* characteristic returned by **t_open()** determines how much data, if any, two users may transfer during connection establishment.
2. the negotiation of protocol options. The client can specify protocol options that would be provided by the transport provider and/or the remote user. XTI supports both local and remote option negotiation.

Note: The option negotiation is inherently a protocol–specific function. This facility is not to be used if protocol-independent software is a goal, except for the XTI_GENERIC Level 0 option.

Warning: TCP/IP does not support sending of data during connection or release phases.

Example of a Client-Program for Connection Establishment in an XTI Connection-oriented Mode Service.

```
        /*---------------------------------------*/
        /* -- ESTB : CONNECTION ESTABLISHMENT -- */
        /*---------------------------------------*/

        if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ALL)) == NULL)
        {
                t_error("t_alloc failed for sndcall");
                exit(6);
        }
        if (info.addr == -1)
        /*      -1 : no limit   ==> field not allocated by t_alloc */
        {
                sndcall->addr.buf = bufaddr;
                sndcall->addr.maxlen = LBUF;
        }
        if (info.options == -1)
        /*      -1 : no limit   ==> field not allocated by t_alloc */
        {
                sndcall->opt.buf = bufopt;
                sndcall->opt.maxlen = LBUF;
        }
        if (t_getraddr(&tp, hostname, servername, &sndcall->addr ) < 0) {
                 /* t_error("t_getraddr failed"); */
                /* New function for Name Service's error */
                t_error_ns("t_getraddr failed");
                exit(7);
        }
        if (trace) {
                fprintf(stdout,"server is:");
                for (i=0, pt=sndcall->addr.buf;
                     i<sndcall->addr.len;
                     i++, pt++)
                        fprintf(stdout, "0x%x,", *pt);
                fprintf(stdout,"\n");
        }
#ifdef XTI4
        if ((ltpdu != FALSE) && ((iso == TRUE) || (rfc == TRUE))) {
                if (t_getopt(&(tp),"example_ltpdu", &sndcall->opt) == -1)
                {
                        t_error_ns ("tgetopt failed");
                        exit (8);
                }
                pt_ltpdu = (unsigned long *)(sndcall->opt.buf + sizeof(struct
t_opthdr));
                *pt_ltpdu = ltpdu;
        }
#else
        if ((ltpdu != FALSE) && ((iso == TRUE) || (rfc == TRUE))) {
                optiso = (struct isoco_options *)sndcall->opt.buf;
                initopt(optiso);
                optiso->mngmt.dflt = T_NO;
                optiso->mngmt.ltpdu = ltpdu;
                sndcall->opt.len = sizeof(struct isoco_options);
        }
#endif
```

```
if (t_connect(fd, sndcall, NULL) < 0) {
        t_error("t_connect failed for fd");
        if (t_errno == TLOOK) {
                if (t_rcvdis(fd, &discon) < 0) {
                        t_error("t_rcvdis failed for fd");
                        exit(9);
                }
                fprintf(stderr, "T_DISCONNECT reason: %x\n",
                        discon.reason);
        }
        exit(9);
}
if (t_free((char *)sndcall, T_CALL) < 0) {
        t_error("t_free failed for sndcall");
        exit(10);
}
```

The **t_connect()** call establishes the connection with the server. The first parameter (**integer**) of **t_connect()** identifies the transport endpoint through which the connection is established. The second parameter identifies the destination server and the third parameter is used to return information to the user. Second and third parameters are specified as a pointer to the **t_call** structure, which has the following format:

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

where:

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

t_alloc() is called to allocate the second parameter **t_call** structure dynamically, before the **t_connect()** call.

Here, the third parameter is set to NULL to indicate that the client does not matter with this information.

The server address is assigned by using the **t_getraddr()** function. Given the specific transport provider as first parameter, a string identifying the server's host as second parameter, a string identifying the server's name application as third parameter, **t_getraddr()** returns the server's protocol address in a netbuf structure as fourth parameter.

The structure of a protocol address on a **t_connect()**, **t_listen()**, **t_rcvconnect()**, **t_getrname()** or **t_getraddr()** depends on the used transport provider.

t_getladdr(), **t_getraddr()**, **t_getlname()**, **t_getrname()** are functions of the XTI Name Server, added to the standard XTI library to facilitate manipulation of protocol addresses. Refer to XTI Name Server Functions, on page 5-2, to have more details on these functions and the use of the associated **xti_ns** library.

- For TCP/IP the XTI Name Service functions use INET Name Service.
- For OSI the XTI Name Service functions access the database files: /**etc**/**xtihosts**, for the host addresses, and /**etc**/**xtiservices** for the application names and the selectors.

For more details about Bull OSI Connection Oriented transport addressing, see “Addressing Concepts” in *OSI Services Reference Manual*.

Note: As the option negotiation is inherently a protocol–specific function, this facility is not to be used if protocol-independent software is a goal, except for the XTI_GENERIC Level 0 option.

Nevertheless, this client example shows how to negotiate options in an OSI Connection case. **t_getopt()** is used to set the option buffer within the **t_call** structure.

In this example, no user data are associated with the **t_connect()** call.

The third parameter of **t_connect()** is used to return information on the newly established connection to the user. It may contain any user data sent by the server with his connect response. It is set to NULL by the client to indicate that this information doesn’t matter to him.

The connection is established on successful return of **t_connect()**. If the server rejects the connect request, **t_connect()** fails and sets **t_errno** to TLOOK. Refer to *X/Open Transport Interface XPG4 CAE Specification Version 2* for more details about TLOOK error.

The Server

When the client calls **t_connect()**, a connect indication is generated on the listening transport endpoint of the server. For each client, the server accepts the connect request and spawns a server process to manage the connection on the responding endpoint.

Endpoint management between the server and the transport provider is described in the illustration.

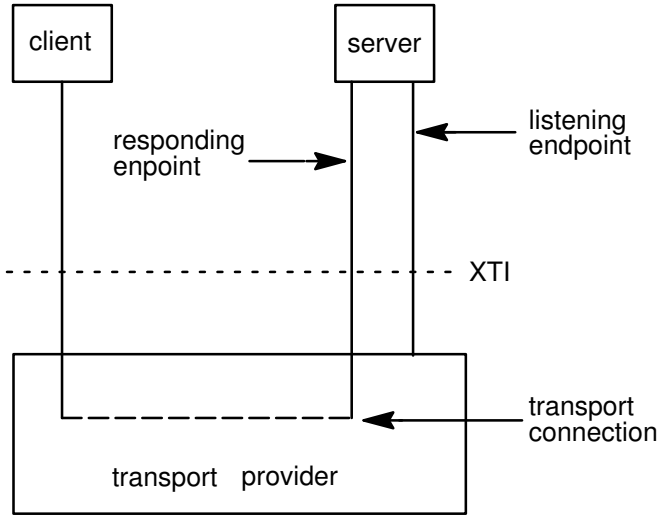


Figure 8. Listening and Responding Transport Endpoints

Connection establishment is managed in two phases:

1. Management of client connection requests,
2. Management of request acceptance

Management of Client Connection Requests (1rst Phase)

Example of a Server-Program for Connection Establishment (1rst Phase) in an XTI Connection-oriented Mode Service.

```
/*-----------------------------------------------------*/
/* -- ESTB1 : CONNECTION ESTABLISHMENT first phase -- */
/*-----------------------------------------------------*/

if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL, T_ALL))
                  == NULL)
{
        t_error("t_alloc failed for call");
        exit(10);
}
if (info.addr == T_INFINITE)
/* -1  : no limit       ==> field not alocated by t_alloc */
{
        call->addr.buf          = bufaddr;
        call->addr.maxlen       = LBUF;
}
if (info.options == T_INFINITE)
/* -1  : no limit       ==> field not alocated by t_alloc */
{
        call->opt.buf           = bufopt;
        call->opt.maxlen        = LBUF;
```

```
}
while (1) {
        if (t_listen(listen_fd, call) < 0) {
                t_error("t_listen failed for listen_fd");
                exit(11);
        }
        if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT) {
                run_server(listen_fd, infile);
        }
/* t_free of bind and call structures be effectively */
/* realised when the server process will be killed */
/* (t_alloc use the general memory allocation function) */
}
```

The server allocates a **t_call** structure to be used by **t_listen()**. The third parameter of **t_alloc()**, set to T_ALL, specifies that all necessary buffers should be allocated for retrieving the caller address, options, and user data.

- As TCP/IP does not support the transfer of user data during connection establishment, and does not support either any protocol options, **t_alloc()** does not allocate buffers for the user data and options.
- With OSI **t_alloc()** allocates buffers for the user data and options, determining the size of the buffers from the *connect* or *disconnect* (respectively the *options*) characteristics returned by **t_open()**.

Furthermore **t_alloc()** must allocate a buffer large enough to store the address of the connection initiator. **t_alloc()** determines the buffer size from the *addr* characteristic returned by **t_open()**. The *maxlen* field of each **netbuf** structure is set to the size of the newly allocated buffer by **t_alloc()**.

- For OSI
  - *maxlen* = 64 for the user data buffer
  - *maxlen* = 2000 for option buffers
- For TCP
  - *maxlen* = 0 for user data and option buffers.
  - *maxlen* = 512 for option buffers

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

Note: For such routines, XTI supports an asynchronous mode which prevents a process from blocking.

Management of Request Acceptance (2nd Phase)

When a connect indication arrives, the server calls **accept_call** to accept the client request.

Description of the **accept_call** function.

```
        /*-------------------------------------------------------*/
        /* -- ESTB2 : CONNECTION ESTABLISHMENT : second phase ----*/
        /*-------------------------------------------------------*/

accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
int resfd;
        if ((resfd = t_open(tp.tp_name, O_RDWR, NULL)) < 0) {
                t_error("t_open for responding fd failed");
                exit(12);
        }
#ifndef XTI4
        if ( iso == TRUE ) {
                bindret->qlen = 0;
                if (t_bind(resfd, bindret, NULL) < 0) {
                        t_error("t_bind for responding fd failed");
                        exit(13);
                }
        }
        /* for TCP the address is not necessary */
        else {
                 if (t_bind(resfd, NULL, NULL) < 0) {
                        t_error("t_bind for responding fd failed");
                        exit(13);
                }
        }
#endif
        /*call->sequence is actually initialized */
        /*because returned by t_listen*/
        call->addr.len = 0;
        call->opt.len = 0;
        call->udata.len = 0;
        if (t_accept(listen_fd, resfd, call) < 0) {
                if (t_errno == TLOOK) {
                        if (t_rcvdis(listen_fd , NULL) < 0) {
                                t_error("t_rcvdis failed for listen_fd");
                                exit(14);
                        }
                        if (t_close(resfd) < 0) {
                                t_error("t_close failed for responding fd");
                                exit(15);
                        }
                        return(DISCONNECT);
                }
                t_error("t_accept failed");
                exit(16);
        }
        return(resfd);
}
```

accept_call accepts the connection on an alternate transport endpoint and returns the value of that endpoint. *conn_fd* is a global variable that identifies the transport endpoint where the connection is established. Because the connection is accepted on an alternate endpoint, the server can continue listening for connect indications on the endpoint that was bound for listening. If the call is accepted without error, **run_server**, described on page 7-27, spawns a process to manage the connection.

accept_call uses two parameters: *listen_fd* identifies the transport endpoint where the connect indication arrived, and *call* is a pointer to a **t_call** structure that contains all information associated with the connect indication. The server first establishes another transport endpoint by opening and binding an address. The newly established transport endpoint, *resfd*, is used to accept the client connect request.

The first two parameters of **t_accept()** function, *listen_fd* and *resfd*, specify the listening transport endpoint where the connect indication arrives and the endpoint where the connection may be accepted, respectively.

Note: A connection can be accepted on the listening endpoint. However, this would prevent other clients from accessing the server for the duration of that connection.

The third parameter of **t_accept()** points to a **t_call** structure. This structure must contain the sequence number identifying the connect indication, returned to by **t_listen()**. Also, the **t_call** structure should identify protocol options the user would like to specify, and user data that may be passed to the client.

To make the example easier, the server exits if either the **t_open()** or **t_bind()** call fails. The **exit()** function closes the transport endpoint associated with *listen_fd*, causing the transport provider to pass a disconnect indication to the client that requested the connection.

- On OSI, this disconnect indication notifies the client that the connection was not established; **t_connect()** fails, setting **t_errno** to TLOOK.
- On TCP, a subsequent XTI routine will fail, setting **t_errno** to TLOOK.

t_accept() can fail if an asynchronous event has occurred on the listening transport endpoint before the connection is accepted, and **t_errno** is set to TLOOK. The two events which may cause **t_accept()** function to return with a TLOOK error are: T_DISCONNECT and T_LISTEN. Because of *qlen*'s values on the previous **t_bind()** (i.e. qlen=1), the T_LISTEN event can not occur in the example. So a disconnect indication should arrived. This event may occur if the client decides to undo the connect request it had previously initiated.

The server must retrieve the disconnect indication using **t_rcvdis()**. This routine takes a pointer to a **t_discon** structure as second parameter, which is used to retrieve information associated with a disconnect indication. In this example, however, the server does not care to retrieve this information, so it sets the parameter to NULL. After receiving the disconnect indication, *accept_call* closes the responding transport endpoint and returns DISCONNECT, which informs the server that the connection was disconnected by the client. The server then listens for further connect indications.

On successful return from **t_accept()**, the transport connection is established on the newly created responding endpoint, and the listening endpoint is freed to retrieve further connect indications.

Data Transfer in an XTI Connection-oriented Mode Service

Once the connection established, both client and server can begin transferring data over the connection.

The following table summarizes the XTI routines which support data transfer operations.

Functions	Description
t_rcv	Retrieves data which has arrived over a transport connection
t_snd	Sends data over an established transport connection

XTI does not differentiate the client from the server in data transfer. Either user can send and receive data, or release the connection. XTI guarantees reliable, sequenced delivery of data over an existing connection.

Two classes of data can be transferred over a transport connection:

- normal data,
- expedited data.

Expedited data is typically associated with information of an urgent nature. The exact semantics of expedited data are subject to the interpretations of the transport provider. Furthermore, all transport protocols do not support the concept of expedited data. See **t_open**() for more details about the *etsdu* field of the **t_info** structure.

- TCP/IP supports the transfer of data in byte stream mode. Byte stream implies no message boundaries on data which are transferred over a connection.
- OSI supports the preservation of message boundaries over a transport connection. The messages, called Transport Service Data Units (TSDU), can be transferred between two transport users as distinct units. The maximum size of a TSDU is a characteristic of the underlying transport protocol. This information is available to the user from **t_open()** and **t_getinfo()**. Because the maximum TSDU size can be large (possibly unlimited), XTI enables a user to transmit a message in multiple units.

  To send a message in smallest units over a transport connection, the user must set the T_MORE flag on every **t_snd()** call except the last one:

  – T_MORE set in a message indicates that the user will send more data associated with the message in a subsequent call to **t_snd()**,

  – T_MORE reset in the last **t_snd()** indicates that this is the end of the TSDU.

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

Note: The T_MORE flag implies nothing about how the data may be packaged under the X/OPEN Transport Interface. Furthermore, it implies nothing about how the data may be delivered to the remote user. Each transport protocol, and each implementation of that protocol, can package and deliver the data differently.
For example, if a user sends a complete message in a single call to **t_snd()**, there is no guarantee that the transport provider will deliver the data in a single unit to the remote transport user. Similarly, a TSDU transmitted in two message units may be delivered in a single unit to the remote transport user. The message boundaries may only be preserved by noting the value of the T_MORE flag on **t_snd()** and **t_rcv()**. This will guarantee that the receiving user will see a message with the same contents and message boundaries as was sent by the remote user.

The Server

Example of a Server-Program for Data Transfer in an XTI Connection-oriented Mode Service.

```
/*--------------------------------*/
/*   -- XFER : DATA TRANSFER --   */
/*--------------------------------*/

run_server(listen_fd, infile)
int listen_fd;
char *infile;
{
        int nbytes;
        FILE *infp;
        char buf[LBUF_MAX];
        char lgth_file[LSIZE];
        switch (fork()) {
        case -1:
                perror("fork failed");
                exit(20);
        default: /* parent */
                if (t_close(conn_fd) < 0) {
                        t_error("t_close failed for conn_fd");
                        exit(21);
                }
                break;
        case 0: /* child */
                if (t_close(listen_fd) < 0) {
                        t_error("t_close failed for listen_fd");
                        exit(22);
                }
                if ((infp = fopen(infile, "r")) == NULL) {
                        perror("cannot open input file");
                        exit(23);
                }
                sprintf(lgth_file,"%d",filesize(infp));
                fprintf(stdout,
                        "server : %d bytes in input file \"%s\" to send\n",
                        filesize(infp), infile);
                if (t_snd(conn_fd, lgth_file, LSIZE, 0) < 0) {
                        if (t_errno == TLOOK) {
                                fprintf(stderr,
                                        "t_snd : connection aborted\n");
                                exit(25);
                        }
                        t_error("t_snd failed for conn_fd");
                        exit(26);
                }
                while ((nbytes = fread(buf, 1 , lbuf, infp)) > 0) {
                        if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
                                if (t_errno == TLOOK) {
                                        fprintf(stderr,
                                                "t_snd : connection aborted\n");
                                        exit(27);
                                }
                                t_error("t_snd failed for conn_fd");
                                exit(28);
                        }
                }
                /* wait  release or disconnection indication */
                connrelease();
        } /*end switch fork*/
}
```

After the **fork()** call, the main process returns to the infinite processing loop and listen for further connect indications. Meanwhile, the secondary process manages the newly established transport connection. If the **fork()** call fails, **exit()** closes the transport endpoint associated with *listen_fd*. This action will cause a disconnect indication to be passed to the client.

The server first sends over the connection in a fixed number of bytes LSIZE, the size of the input file it will transfer later.

The server process then reads a number of bytes= *lbuf* of the input file at a time and sends these data to the client using **t_snd()**. *buf* points to the start of the data buffer, and *nbytes* specifies the number of bytes to be transmitted. The fourth parameter *flags* is used to specify options. It can be set to T_EXPEDITED or T_MORE, or both. Neither flag is set by the server in this example.

If the user begins to flood the transport provider with data, the XTI library provides flow control. In such cases, **t_snd()** fails and **t_errno** is set to **TFLOW.** To resume the data transfer operation, the user must scan the flow state, waiting for a T_GODATA or T_GOEXDATA return value of **t_look()**, and then call the **t_snd()** function again.

Example of a Client-Program for Data Transfer in an XTI Connection-oriented Mode Service.

The client receives the data transfered by the server over the transport connection, then writes this data to its standard output file.

```
/*---------------------------------------*/

/* -- XFER : DATA TRANSFER -------------- */
/*---------------------------------------*/

if ((outfp = fopen(outfile, "w")) == NULL) {
        perror("cannot open output file");
        exit(11);
}
/* first data received is the size of transferred file */
if ((nbytes = t_rcv(fd, buf, LSIZE, &flags)) < 0) {
        if (t_errno == TLOOK) {
                if ((evt = t_look(fd)) < 0) {
                        t_error("t_look failed for fd");
                        exit(12);
                }
                fprintf(stderr, "t_rcv evt: %x\n", evt);
                exit(13);
        }
        t_error("t_rcv failed for fd");
        exit(14);
}
sscanf(buf, "%d", &lgth_file);
fprintf(stdout, "client : %d bytes in file to receive\n",
        lgth_file);
while (rcv_bytes < lgth_file) {
        if ((nbytes = t_rcv(fd, buf, LBUF_MAX, &flags)) < 0) {
                if (t_errno == TLOOK) {
                        if ((evt = t_look(fd)) < 0) {
                                t_error("t_look failed for fd");
                                exit(15);
                        }
                        fprintf(stderr, "t_rcv evt: %x\n", evt);
                        exit(16);
                }
                t_error("t_rcv failed for fd");
                exit(17);
        }
        if (fwrite(buf, 1, nbytes, outfp) < 0) {
                fprintf(stderr, "fwrite failed\n");
                exit(18);
        }
        rcv_bytes = rcv_bytes + nbytes;
}
fprintf(stdout,
        "client : %d bytes received in output file \"%s\"\n",
        rcv_bytes,outfile);
```

The client continuously calls **t_rcv()** to process incoming data. If no data is currently available, **t_rcv()** blocks until data arrives. **t_rcv()** retrieves the available data up to LBUF_MAX bytes, which is the size of the client's input buffer, and returns the number of received bytes. Then, the client writes these data to standard output and continues. The data transfer phase is completed when the length of the sent file is reached or when **t_rcv()** fails. **t_rcv()** may fail if an orderly release indication or disconnect indication arrives, as discussed in Connection Release, on page 7-31. If the **fwrite()** call fails for any reason, the client will exit, thereby closing the transport endpoint. If the transport endpoint is closed (either by **exit()** or **t_close()**) when it is in the data transfer phase, the connection will be aborted and the remote user will receive a disconnect indication.

Connection Release in an XTI Connection-oriented Mode Service

At any point during data transfer, any user can release the transport connection and end the conversation. Two forms of connection release are supported by XTI:

1. Abortive release, which breaks a connection immediately and can result in the loss of any data that has not yet reached the destination user. An abortive release may be generated by any user using the **t_snddis()** function or directly by the transport provider if a problem occurs below XTI.
   All transport providers support the abortive release.

   t_snddis() function enables a user to send data to the remote user when aborting a connection. This facility is not supported by all transport providers.

   When the remote user is notified of the aborted connection, the **t_rcvdis()** function must be called to retrieve the disconnect indication. **t_rcvdis()** returns a code that indicates why the connection was aborted, and returns any user data that may have accompanied the disconnect indication (if the abortive release was initiated by the remote user).

Note: As the errror code is specific to the underlying transport protocol, it must not be interpreted by protocol–independent software.

2. Orderly release, which terminates a connection without loss of data**,** using the **t_sndrel()** and **t_rcvrel()** functions.
   Orderly release is an optional facility that is not supported by all transport protocols. For example, OSI does not support it.

To avoid loss of the end of file data during an abortive release, the connection release cannot be initiated by the server. Therefore, it is the client's responsibility to initiate the connection release when all the data from the transferred file has been received. Many other mechanisms are possible for preventing data loss.

The following table summarizes the XTI routines which support connection release.

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

The Client

The two forms of connection release are described in this example:

- abortive release with OSI,
- orderly release with TCP/IP.

```
/*--------------------------------------*/
/* -- REL : CONNECTION RELEASE */
/*--------------------------------------*/

if (info.servtype == T_COTS_ORD) {
        if (t_sndrel(fd) < 0) {
                t_error("t_sndrel failed for fd");
                exit(19);
        }
        fprintf(stdout,
                "client : orderly release initiated\n");
        while (t_rcvrel(fd) < 0) {
                /* If the event is TNOREL : it's OK , we must wait for
                   the orderly release . Else , it's an error */
                if (t_errno != TNOREL) {
                        if ((evt = t_look(fd)) < 0) {
                                t_error("t_look failed for fd");
                                exit(20);
                        }
                        fprintf(stderr,
                                "t_rcvrel evt: %x\n", evt);
                }
                if (t_errno != TNOREL) {
                        t_error("t_rcvrel failed for fd");
                        exit(21);
                }
        }
        fprintf(stdout,
                "client : orderly release completed ok\n");
}
else {
        if (t_snddis(fd, NULL) < 0) {
                t_error("t_snddis failed for fd");
                exit(22);
        }
        fprintf(stdout,
                "client : abortive disconnection requested ok\n");
}
exit(0);
```

The abortive release procedure is initiated by calling the **t_snddis()** function.

The orderly release procedure consists of two steps performed by each user. One user may initiate a release using **t_sndrel()**. This routine informs the other user that no more data will be sent. When this other user receives the orderly release indication, he can go on sending data, if desired. When he has no more data to send, it's his turn to call **t_sndrel()** to indicate that he is ready to release the connection. The connection is released only after both users have requested an orderly release and received the corresponding indication from the other user.

The Server

The two forms of connection release are described in this example:

```
/*--------------------------------*/
/* -- REL : CONNECTION RELEASE -- */
/*--------------------------------*/

void connrelease()
{
int evt;
        /* until release or disconnection indication arrives */
        while ((evt = t_look(conn_fd)) == 0) {
                sleep(1);
        }
        if (evt < 0) {
                t_error(”t_look failed for conn_fd”);
                exit(30);
        }
        switch (evt) {
        case T_DISCONNECT:
                if (t_rcvdis(conn_fd, NULL) < 0 ){
                    t_error(”t_rcvdis failed”);
                    exit(31);
                }
                fprintf(stdout,
                        ”server : abortive disconnection received ok\n”);
                exit(0);
        case  T_ORDREL:
                if (t_rcvrel(conn_fd) < 0 ){
                    t_error(”t_rcvrel failed”);
                    exit(31);
                }
                fprintf(stdout, ”server : orderly release received\n”);
                if (t_sndrel(conn_fd) < 0 ){
                    t_error(”t_sndrel failed”);
                    exit(32);
                }
                fprintf(stdout,
                        ”server : orderly release acknowledged ok\n”);
                exit(0);
        default:
                fprintf(stderr, ”server : error evt: %x\n”, evt);
                exit(31);
        }
}
```

The server calls **connrelease()** when it has finished sending the input file. **connrelease()** manages the connection release phase. The server waits incoming events with **t_look()**. A T_ORDREL indication (for TCP) or a T_DISCONNECT indication (for ISO) is received after the client has initiated the release.

If a T_ORDREL indication is received, the server calls **t_rcvrel()** to process this indication and calls **t_sndrel()**. The connection is released.

If a T_DISCONNECT indication is received, the server may call **t_rcvdis()** to process this indication.

The exit call in **connrelease()** closes the transport endpoint. If a user process wants to close a transport endpoint without existing it may call **t_close()**.

Overview of an XTI Connectionless Mode Service

Connectionless Mode is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. These units are also known as datagrams. This service:

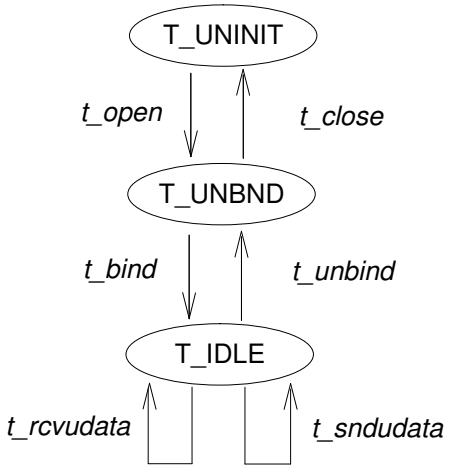
- requires a pre-existing association between the peer users involved, which determines the characteristics of the data to be transmitted,
- does not enable the dynamic negotiation of parameters and options.

All the information required to deliver a unit of data (for example, the destination address) is presented to the transport provider, together with the data to be transmitted, in one service access. Each unit of data transmitted is entirely self-contained and can be independently routed by the transport provider.

This Connectionless Mode service is attractive for applications which:

- involve short-term request/response interactions,
- exhibit a high level of redundancy,
- can be dynamically reconfigured,
- do not require guaranteed, in-sequence delivery of data.

The example of Connectionless Mode service described in this state machine is not meant to show all the functions that may be called, but rather to hightlight the main functions that requests a particular service request



	User A	User B
Local Management : Initialisation	t_open() t_bind()	t_open() t_bind()
Data Transfer	t_sndudata() t_rcvudata()	t_rcvudata() t_sndudata()
Local Management : De-initialisation	t_unbind() t_close()	t_unbind() t_close()

Figure 9. Sequence of XTI functions in Connectionless Mode

The Connectionless Mode is well adapted to transaction processing applications. So a Connectionless Mode service is described using a transaction server as example. This server waits for incoming transaction queries, and processes and responds to each query.

A Connectionless Mode server is described step by step:

- Local Management, on page 7-36, to establish a communication channel with the Transport Provider.
- Data Transfer, on page 7-38, to exchange datagrams over the Transport Provider endpoint,
- Datagram Errors, on page 7-40, returned by the Transport Provider.

The example is described using Name Server.

The directory */**usr/lpp/xti_api/examples/binop_c/clts** contains the source files of the complete programs:

clts_client.c client-program of an XTI Connectionless Mode service using Name Server

clts_server.c server-program of an XTI Connectionless Mode service using Name Server

Local Management in an XTI Connectionless Mode Service

The local management phase uses the same local operations as those described in Connection-oriented Mode Local Management, on page 7-10.

The user must choose the appropriate connectionless transport provider using **t_open()** and establish its identity using **t_bind()**.

Example of a Server-Program for Local Management in an XTI Connectionless Mode Service.

```
/*--------------------------------*/
/* -- LMGMT : LOCAL MANAGEMENT -- */
/*--------------------------------*/

if (osi_cltp)
        tp.tp_id = TPID_OSI_CLTS;
else
        tp.tp_id = TPID_UDP;

if (t_gettp(&tp) < 0) {
        /* New function for Name Service's errors */
        t_error_ns("t_gettp failed");
        exit(2);
}
if ((fd = t_open(tp.tp_name, O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(3);
}

if ((bind = (struct t_bind *)(t_alloc(fd, T_BIND, T_ALL)))
          == NULL) {
        t_error("t_alloc failed for bind");
        exit(4);
}
if (t_getladdr(&tp, servername, &bind->addr) < 0) {
        /* New function for Name Service's errors */
        t_error_ns("t_getladdr failed");
        exit(5);
}
/* bind->qlen = 0; */
if ((bindret = (struct t_bind *)(t_alloc(fd, T_BIND, T_ALL)))
             == NULL) {
        t_error("t_alloc failed for bindret");
        exit(6);
}
if (t_bind(fd, bind, bindret) < 0) {
        t_error("t_bind failed");
        exit(7);
}
```

The server establishes a transport endpoint with the desired transport provider using **t_open()**. Each provider has an associated service type, so the user may choose a particular service by opening the appropriate transport provider file. The path name corresponding to UDP is: */dev/xti/udp*. The path name corresponding to OSI CLTS is:

/dev/xti/cltp. This connectionless–mode server ignores the characteristics of the provider returned by **t_open()** in the same way as the users setting the third argument to NULL in the connection–mode example. For simplicity, the transaction server assumes the transport provider has the following characteristics:

- The transport provider supports the T_CLTS service type (connectionless transport service, or datagram).
- The transport provider does not support any protocol–specific options.

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

The *qlen* field of the **t_bind** structure is insignificant for connectionless–mode service, because all users can receive datagrams once they have been bound to an address. XTI defines an inherent client–server relationship between two users while establishing a transport connection in the connection–mode service. However, no such relationship exists in the connectionless–mode service. It is the context of this example, and not the X/OPEN Transport Interface rules, which defines one user as a server and another as a client.

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

Data Transfer in an XTI Connectionless Mode Service

Once a user has been bound to an address to the transport endpoint, datagrams may be sent or received over that endpoint using the **t_sndudata**() and **t_rcvudata**() routines. Each outgoing message is accompanied by the transport address of the destination user. In addition, XTI enables a user to specify protocol options that should be associated with the transfer of the data unit (for example, transit delay). As discussed in Local Management, each transport provider defines the set of options which may be possibly associated to a datagram. When the datagram is passed to the destination user, the associated protocol options may be returned as well. UDP does not support options.

Example of a Server-Program for Data Transfer including Datagrams Errors management in an XTI Connectionless Mode Service.

```
/*--------------------------------------*/
/* -- XFER : DATA TRANSFER -- */
/*--------------------------------------*/

if ((ud = (struct t_unitdata *)(t_alloc(fd, T_UNITDATA, T_ALL)))
        == NULL) {
        t_error("t_alloc failed for t_unitdata structure");
        exit(8);
}
if ((uderr = (struct t_uderr *)(t_alloc(fd, T_UDERROR, T_ALL)))
           == NULL) {
        t_error("t_alloc failed for t_uderr structure");
        exit(9);
}
for (i=0; ; i++) {
        if (t_rcvudata(fd, ud, &flags) < 0) {
                if (t_errno == TLOOK) {
                        /* error on previously sent datagram */
                        if (t_rcvuderr(fd, uderr) < 0) {
                                t_error("t_rcvuderr failed");
                                exit (10);
                        }
                        fprintf(stderr,"bad datagram : error = %d\n",
                                uderr->error);
                        continue;
                }
                t_error("t_rcvudata failed");
                fprintf(stdout,"server: %d datagrams processed\n");
                exit (11);
        }
        if (trace) {
                fprintf(stdout,"R");
                fflush(stdout);
        }
        alarm(ALRM_DELAY);
        /*
         * query() processes the request and places the
         * response in ud->udata.buf, setting ud->udata.len
         */
        query(ud);
        if (t_sndudata(fd, ud) < 0) {
                t_error("t_sndudata failed");
                exit (12);
        }
```

```
                if (trace) {
                        fprintf(stdout,"S");
                        fflush(stdout);
                }
        }
}
```

To store datagrams, the server must first allocate a **t_unitdata** structure:

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

- the *addr* field holds the source address of incoming datagrams and the destination address of outgoing datagrams,
- the *opt* field identifies any protocol options associated with the transfer of the datagram,
- the *udata* field holds the data itself.

The *addr*, *opt*, and *udata* fields must all be allocated with buffers which are large enough to hold any possible incoming values. As described in Connection Establishment in an XTI Connection-oriented Mode Service, on page 7-17, this will be ensured by the T_ALL value, given to **t_alloc()** as third parameter. The *maxlen* field of each **netbuf** structure is set accordingly.

Because UDP provider does not support protocol options in this example, no options buffer is allocated, and *maxlen* is set to zero in the **netbuf** structure for options. A **t_uderr** structure is also allocated by the server for processing any datagram errors, as discussed in Datagram errors, on page 7-40.

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

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

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

When the response is ready, **t_sndudata()** is called to return the response to the client. XTI prevents a user from flooding the transport provider with datagrams using the same flow control mechanism as the one described for the Data Transfer in a Connection-oriented Mode Service, on page 7-28. In such cases, to resume the data transfer operation, the user must scan the flow state, waiting for the flow control to be relieved, and then call the **t_sndudata()** function again.

Datagram Errors in an XTI Connectionless Mode Service

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

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

The transaction server will be notified of this error event when it attempts to receive another datagram. In this case, **t_rcvudata()** will fail, setting **t_errno** to TLOOK. If TLOOK is set, the only possible event is T_UDERR, so the server calls **t_rcvuderr()** to retrieve the event. The second parameter of **t_rcvuderr()** is the **t_uderr** structure, allocated at the beginning of Data Transfer example, on page 7-38. This structure, filled in by **t_rcvuderr(),** has the following format define in **<xti.h>**:

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

- the *addr* field identifies the destination address,
- the *opt* field identifies the protocol options as specified in the bad datagram,
- the *error* field is a protocol–specific error code that indicates why the provider could not process the datagram.

The transaction server prints the error code and then continues by entering the processing loop again.

Example of Read/Write Interface for XTI Applications

Purpose

A user may wish to establish a transport connection and then **exec**() an existing program such as **cat**() to process the data as it arrives over the connection. These existing programs use **read**() and **write**() for their input/output needs.

XTI does not directly support a **read**/**write** interface to a transport provider, but such an interface, the **tirdwr** module, is provided with the Operating System. This interface enables a user to issue **read** and **write** calls over a transport connection that is in the data transfer phase.

This example describes how to use this **read**/**write** interface in an XTI Connection-oriented Mode service. (This interface is not available in Connectionless Mode.

The **read**/**write** interface is presented using the client-example of a Connection-oriented Mode service, described on page 7-8, with some minor modifications. The program is identical until the data transfer phase is reached. At that point, this client uses the **read**/**write** interface and **cat**() to process incoming data. **cat()** can be run without change over the transport connection.

Only the differences are described:

The directory **/usr/lpp/xti_api/examples/binop_c/cots** contains the source file of the complete program **cots_crdwr.c**.

```
/*
*  .
*  . Same include and define
*  .
*/
#include <stropts.h>

/*
*  .
*  . Same local management and connection
*  . establishment steps.
*  .
*/

    /* -- XFER : DATA TRANSFER -- */

    if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
        perror("I_PUSH of tirdwr failed");
        exit (10);
    }

    close(0);
    dup(fd);

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

    perror("execl of /bin/cat failed");
    exit(11);

    /* -- END -- */
```

The client invokes the **read**/**write** interface by pushing the **tirdwr** module onto the Stream associated with the transport endpoint where the connection was established. For more details, see I_PUSH in **streamio()** function, in *STREAMS Programmer's Guide*. This module converts the X/OPEN Transport Interface above the transport provider into a pure **read**/**write** interface. With the module in place, the client calls **close()** and **dup()** to establish the transport endpoint as its standard input file, and uses /**bin**/**cat** to process the input. Because the transport endpoint identifier is a file descriptor, the facility for duping the endpoint is available to users.

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

CAUTION:
The **tirdwr** module must only be pushed onto a Stream when the transport endpoint is in the data transfer phase. Once the module is pushed, the user must not call any XTI routines. If a XTI routine is invoked, **tirdwr** generates the fatal protocol error EPROTO on the Stream, making it unusable. Furthermore, if the user pops the **tirdwr** module off the Stream, the transport connection will be aborted. For more details about processing **tirdwr** module, see I_POP in **streamio(7)** manual pages of *STREAMS Programmer's Guide*.

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

Write

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

Read

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

- **read()** cannot process expedited data because it cannot distinguish expedited data from normal data for the user. If an expedited data indication is received, **tirdwr** generates a fatal protocol error, EPROTO, on that Stream. This error causes further system calls to fail. Therefore, the user must be aware that he must not communicate with a process which is sending expedited data.
- If an abortive disconnect indication is received, **tirdwr** discards the indication and generates a STREAMS hang–up condition on that Stream. Subsequent **read()** calls retrieves any remaining data, and then **read()** returns zero for all further calls (indicating end–of–file).
- If an orderly release indication is received, **tirdwr** discards the indication and delivers a zero–length STREAMS message to the user. As described in **read(),** this notifies the user of end–of–file by returning 0 to the user.

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

Close

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

- If an orderly release indication has been previously received by **tirdwr**, an orderly release is requested to the transport provider to complete the orderly release of the connection. The remote user, who initiated the orderly release procedure, will receive the expected indication when data transfer completes.
- If a disconnect indication had previously been received by **tirdwr**, no special action is processed.
- If neither an orderly release indication nor disconnect indication has been previously received by **tirdwr**, a disconnect is requested to the transport provider to abortively release the connection.
- If an error had previously occurred on the Stream and a disconnect indication has not been received by **tirdwr**, a disconnect is requested to the transport provider.

A process cannot initiate an orderly release after **tirdwr** is pushed onto a Stream, but **tirdwr** properly handles an orderly release if it is initiated by the user on the other side of a transport connection. With TCP, if the client, like the one described in this chapter, communicates with a server program as described in the example in “Data Transfer in an XTI Connection-oriented Mode Service”, on page 7-26, that server should terminate the transfer of data with an orderly release request. The server then waits for the corresponding indication from the client. At that point, the client exits and the transport endpoint is closed. As explained in the first bullet item above, when the file descriptor is closed **tirdwr** initiates the orderly release request from the client’s side of the connection. This request generates the indication expected by the server, and the connection is be released properly.

XTI Program Example using Threads

The following programs represent the same programs as the Connection-oriented Mode Service examples, but using Threads. The thread-specific code is printed in bold.

- Client-example
- Server-example

The Client

```
/*
* @BULL_COPYRIGHT@
*
*/
```

#include <pthread.h>

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <xti3to4.h>
#include <xti.h>
#include <xti_ns.h>
#include <macros.h>

#define LSIZE                 20
#define LTPDU_MIN   T_LTPDUDFLT
#define LTPDU_MAX   8192
#define LBUF_MAX    8192
```

#define MAX_THREAD 100
#define DEFAULT_THREAD 1

```
#define OUTFILE_DFLT          ”outfile”
#define SERVERHOST_DFLT       ”localhost”
#define SERVERNAME_DFLT       ”cots_server”
#define CHECK(status,string)   if (status == -1) perror (string)
```

pthread_mutex_t cond_mutex; /* mutex used for ensuring integrity */
pthread_cond_t cond_var; /* condition variable for thread synchro */
pthread_attr_t pthr_attr_std; /* thread attribute */
int thread_hold = 1; /* number associated with condition state */
struct th_hand {
pthread_t thread_handler;
int status;
int num_th;
} ;
struct th_hand tab_handler[MAX_THREAD]; /* array of client threads */

```
char     *outfile = OUTFILE_DFLT;
char       *hostname = SERVERHOST_DFLT;
int         iso = FALSE;
int        rfc = FALSE;
char       *servername = SERVERNAME_DFLT;
int         ltpdu = FALSE;
int         trace = FALSE;
struct    xtitp tp;
```

int max_thread = DEFAULT_THREAD;

```
extern char *optarg;
extern int getopt();
```

```
usage(type, proc, value)
int type;
char *proc;
char *value;
{
        switch (type) {
        case 1:
                fprintf(stderr, "error on length of TPDU : %s\n", value);
                break;
        case 2:
                fprintf(stderr, "length of TPDU requested ");
                fprintf(stderr, "on non ISO transport provider\n");
                break;
        default:
                break;
        }
          fprintf(stderr,
                "usage: %s [-fF] [-hH] [-i|r] [-nN] [-sS] [-tT] [-v]\n", proc);
        fprintf(stderr,
                "   -fF       set output file name to F ");
        fprintf(stderr,"(default is \"%s\")\n", OUTFILE_DFLT);
        fprintf(stderr,
                "   -hH       set server host name to H ");
        fprintf(stderr,"(default is \"%s\")\n", SERVERHOST_DFLT);
          fprintf(stderr,
                "   -i        set transport provider to ISO ");
        fprintf(stderr,
                "or -r        set transport provider to RFC1006 ");
        fprintf(stderr,"(default is TCP)\n");
          fprintf(stderr,
                    "         -nN       set number of thread to N (default is 1, max is 100)\n");
        fprintf(stderr,
                "   -sS       set server name to S ");
        fprintf(stderr,"(default is \"%s\")\n", SERVERNAME_DFLT);
        fprintf(stderr,
                "   -tT       set length of ISO TPDU to T ");
        fprintf(stderr,"(default is %d, min is %d, max is %d)\n",
                T_LTPDUDFLT, LTPDU_MIN, LTPDU_MAX);
        fprintf(stderr,
                "   -v        set on verbose mode (default is OFF)\n");
}
```

thread_client(void *arg)
{

```
          int fd;
        struct t_info info;
        struct t_call *sndcall;
          struct t_opthdr *optiso;
        unsigned long *pt_ltpdu;
        struct t_discon discon;
          int evt;
          int nbytes;
          FILE *outfp;
          char buf[LBUF_MAX];
          int rcv_bytes = 0;
          int lgth_file = 0;
          int flags = 0;
          int i;
          char *pt;
          char *outfile_prefix = OUTFILE_DFLT;
          char local_outfile[10];
          int status;
```

struct th_hand *pt_handler;
int my_number = *((int *)arg);

```
#define LBUF 250
          char bufopt[LBUF];
        char bufaddr[LBUF];

          if (trace) {
                    fprintf(stdout,"client %d : thread client input\n", my_number);
          }
          fflush(stdout);

          if ((my_number < 0) || (my_number >= MAX_THREAD)) {
                    fprintf(stdout,"client %d : my_number not in range\n",
                              my_number);
                fflush(stdout);
                pthread_exit(NULL);
        }
```

pt_handler = &(tab_handler[my_number]);

pt_handler–>status = my_number;

```
          /* synchronise threads */
          status = pthread_mutex_lock(&cond_mutex);
          CHECK(status,"mutex lock bad status\n");
          while (thread_hold) {
                    status = pthread_cond_wait(&cond_var,&cond_mutex);
                    CHECK(status,"cond_wait bad status\n");
          }
          status = pthread_mutex_unlock(&cond_mutex);
          CHECK(status,"mutex unlock bad status\n");

          if (t_gettp(&tp) < 0) {
                 /* t_error("t_gettp failed"); */
                /* New function for Name Service's error */
                    if (t_error_ns("client : t_gettp failed") < 0)
                              fprintf(stderr,
                              "client %d : t_error_ns failed\n", my_number);
                    pt_handler–>status = –1;
                    pthread_exit((void *)&(pt_handler–>status));
        }

          if ((fd = t_open(tp.tp_name, O_RDWR, &info)) < 0) {
                    fprintf(stderr,
                              "client %d : t_open failed for fd, error : %s\n",
                              my_number, t_strerror(t_errno));
                    pt_handler–>status = –1;
                    pthread_exit((void *)&(pt_handler–>status));
          }

          if (t_bind(fd, NULL, NULL) < 0) {
                    fprintf(stderr,
                              "client %d : t_bind failed for fd, error : %s\n",
                              my_number, t_strerror(t_errno));
                    pt_handler–>status = –1;
                    pthread_exit((void *)&(pt_handler–>status));
          }

          /* -- ESTB : CONNECTION ESTABLISHMENT -- */

          if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL_STR, T_ALL))
                       == NULL) {
                    fprintf(stderr,
                              "client %d : t_alloc failed for sndcall, error : %s\n",
                              my_number, t_strerror(t_errno));
                    pt_handler–>status = –1;
                    pthread_exit((void *)&(pt_handler–>status));
          }
          if (info.addr == -1)
        /*      -1 : no limit   ==> field not allocated by t_alloc      */
```

```
{
        sndcall->addr.buf = bufaddr;
        sndcall->addr.maxlen = LBUF;
}
if (info.options == -1)
/*      -1 : no limit   ==> field not allocated by t_alloc      */
{
        sndcall->opt.buf = bufopt;
        sndcall->opt.maxlen = LBUF;
}
  if (t_getraddr(&tp, hostname, servername, &sndcall->addr ) < 0) {
             /* t_error("t_getraddr failed"); */
            /* New function for Name Service's error */
            if (t_error_ns("client : t_getraddr failed") < 0)
                      fprintf(stderr,
                      "client %d : t_error_ns failed\n", my_number);
            pt_handler->status = -1;
            pthread_exit((void *)&(pt_handler->status));
  }
  if (trace) {
            fprintf(stdout,"client %d : server is:", my_number);
            for (i=0, pt=sndcall->addr.buf;
                 i<sndcall->addr.len;
                 i++, pt++)
                      fprintf(stdout, "0x%x,", *pt);
            fprintf(stdout,"\n");
  }
  if ((ltpdu != FALSE) && ((iso == TRUE) || (rfc == TRUE))) {
        if (t_getopt(&(tp),"example_ltpdu", &sndcall->opt) == -1) {
                      if (t_error_ns("client : t_getopt failed") < 0)
                                fprintf(stderr,
                                "client %d : t_error_ns failed\n", my_number);

                      pt_handler->status = -1;
                      pthread_exit((void *)&(pt_handler->status));
          }
         pt_ltpdu = (unsigned long *)(sndcall->opt.buf +
                                      sizeof(struct t_opthdr));
         *pt_ltpdu = ltpdu;
 }
  if (t_connect(fd, sndcall, NULL) < 0) {
            fprintf(stderr,
                      "client %d : t_connect failed for fd, error : %s\n",
                      my_number, t_strerror(t_errno));
            if (t_errno == TLOOK) {
                      if (t_rcvdis(fd, &discon) < 0) {
                                fprintf(stderr,
                      "client %d : t_rcvdis failed for fd, error : %s\n",
                                      my_number, t_strerror(t_errno));
                                pt_handler->status = -1;
                                pthread_exit((void *)&(pt_handler->status));
                      }
                      fprintf(stderr, "client %d : T_DISCONNECT reason: %x\n",
                                my_number, discon.reason);
            }
            pt_handler->status = -1;
            pthread_exit((void *)&(pt_handler->status));
  }
  if (t_free((char *)sndcall, T_CALL_STR) < 0) {
            fprintf(stderr,
                "client %d : t_free failed for sndcall, error : %s\n",
                my_number, t_strerror(t_errno));
            pt_handler->status = -1;
            pthread_exit((void *)&(pt_handler->status));
```

```
}
/* -- XFER : DATA TRANSFER -- */
sprintf(local_outfile,"%s_%d",outfile_prefix,my_number);
if ((outfp = fopen(local_outfile, "w")) == NULL) {
          perror("cannot open output file");
          pt_handler–>status = –1;
          pthread_exit((void *)&(pt_handler–>status));
}
/* first data received is the size of transferred file */
if ((nbytes = t_rcv(fd, buf, LSIZE, &flags)) < 0) {
          if (t_errno == TLOOK) {
                    if ((evt = t_look(fd)) < 0) {
                              fprintf(stderr,
              "client %d : t_look failed for fd, error : %s\n",
                    my_number, t_strerror(t_errno));
                              pt_handler–>status = –1;
                              pthread_exit((void *)&(pt_handler–>status));
                    }
                    fprintf(stderr, "client %d : t_rcv size evt: %x\n",
                      my_number, evt);
                    pt_handler–>status = –1;
                    pthread_exit((void *)&(pt_handler–>status));
          }
          fprintf(stderr,
              "client %d : t_rcv failed for fd, error : %s\n",
              my_number, t_strerror(t_errno));
          pt_handler–>status = –1;
          pthread_exit((void *)&(pt_handler–>status));
}
sscanf(buf, "%d", &lgth_file);
fprintf(stdout, "client %d : %d bytes in file to receive\n",
        my_number,lgth_file);
while (rcv_bytes < lgth_file) {
          if ((nbytes = t_rcv(fd, buf, LBUF_MAX, &flags)) < 0) {
                    if (t_errno == TLOOK) {
                              if ((evt = t_look(fd)) < 0) {
                                        fprintf(stderr,
                    "client %d : t_look failed for fd, error : %s\n",
                                                  my_number, t_strerror(t_errno));
                                        pt_handler–>status = –1;
                                        pthread_exit(
                                                  (void *)&(pt_handler–>status));
                              }
                              fprintf(stderr, "client %d : t_rcv evt: %x\n",
                              my_number, evt);
                              pt_handler–>status = –1;
                              pthread_exit((void *)&(pt_handler–>status));
                    }
                    fprintf(stderr,
                     "client %d : t_rcv failed for fd, error : %s\n",
                     my_number, t_strerror(t_errno));
                    pt_handler–>status = –1;
                    pthread_exit((void *)&(pt_handler–>status));
          }
          if (fwrite(buf, 1, nbytes, outfp) < 0) {
                    fprintf(stderr, "client %d : fwrite failed\n",
                      my_number);
                    pt_handler–>status = –1;
                    pthread_exit((void *)&(pt_handler–>status));
          }
          rcv_bytes = rcv_bytes + nbytes;
}
```

```
        if (fclose(outfp) < 0)
                fprintf(stderr, "client %d : fclose failed\n", my_number);
        fprintf(stdout,
                "client %d : %d bytes received in output file \"%s\"\n",
                my_number,rcv_bytes,local_outfile);

        /* -- REL : CONNECTION RELEASE */

        if (info.servtype == T_COTS_ORD) {
                if (t_sndrel(fd) < 0) {
                        fprintf(stderr,
                            "client %d : t_sndrel failed for fd, error : %s\n",
                                my_number, t_strerror(t_errno));
                        pt_handler->status = -1;
                        pthread_exit((void *)&(pt_handler->status));
                }
                fprintf(stdout,
                        "client %d : orderly release initiated\n",my_number);
                while (t_rcvrel(fd) < 0) {
                        /* If the event is TNOREL : it's OK , we must wait for
                           the orderly release . Else , it's an error */
                        if (t_errno != TNOREL) {
                                if ((evt = t_look(fd)) < 0) {
                                        fprintf(stderr,
                                    "client %d : t_look failed for fd, error : %s\n",
                                                my_number, t_strerror(t_errno));
                                                pt_handler->status = -1;
                                                pthread_exit(
                                                        (void *)&(pt_handler->status));
                                }
                                fprintf(stderr,
                                                "client %d : t_rcvrel evt: %x\n",
                                                my_number, evt);
                        }
                        if (t_errno != TNOREL) {
                                fprintf(stderr,
                        "client %d : t_rcvrel failed for fd, error : %s\n",
                                my_number, t_strerror(t_errno));
                                pt_handler->status = -1;
                                pthread_exit((void *)&(pt_handler->status));
                        }

                }
                fprintf(stdout,
                        "client %d : orderly release completed ok\n",my_number);
        }
        else {
                if (t_snddis(fd, NULL) < 0) {
                        fprintf(stderr,
                        "client %d : t_snddis failed for fd, error : %s\n",
                           my_number, t_strerror(t_errno));
                        pt_handler->status = -1;
                        pthread_exit((void *)&(pt_handler->status));
                }
                fprintf(stdout,
                        "client %d : abortive disconnection requested ok\n",
                          my_number);
        }
        if (trace) {
                fprintf(stdout, "client %d : thread output \n",my_number);
      }
        pthread_exit((void *)&(pt_handler->status));
}
```

```
/*********************************************/
/*                 COTS_CLIENT_R                          */
/*********************************************/

main(argc,argv)
int argc;
char *argv[];
{
        int c;
          int receiver_num;
          int exit_value;
          int status;
          int pt_status;

          while ((c = getopt(argc,argv,"f:h:irs:t:vn:"))
                  != -1) {
                switch (c) {
                case 'f':
                        outfile = optarg;
                        break;
                case 'h':
                        hostname = optarg;
                        break;
                    case 'i':
                        if (rfc==FALSE)
                                iso = TRUE;
                        else {
                                usage(-1, argv[0], NULL);
                                        pthread_exit((void *) NULL);
                        }
                        break;
                case 'r':
                        if (iso==FALSE)
                                rfc = TRUE;
                        else {
                                usage(-1, argv[0], NULL);
                                        pthread_exit((void *) NULL);
                        }
                        break;
                    case 'n':
                              max_thread = max(DEFAULT_THREAD,
                                                      min(atoi(optarg),MAX_THREAD));
                              break;
                case 's':
                        servername = optarg;
                        break;
                case 't':
                        ltpdu = atoi(optarg);
                        if (ltpdu < LTPDU_MIN || ltpdu > LTPDU_MAX) {
                                usage(1, argv[0], optarg);
                                        pthread_exit((void *) NULL);
                        }
                        break;
                case 'v':
                        trace = TRUE;
                        break;
                case 'n':

                default:
                            usage(-1, argv[0], NULL);
                              pthread_exit((void *) NULL);
                }
        }
          if ((ltpdu != FALSE) && ((iso != TRUE) && (rfc != TRUE))) {
                usage(2, argv[0], NULL);
```

```
                pthread_exit((void *) NULL);
        }
        if (iso == TRUE) {
                tp.tp_id = TPID_OSI_COTS;
        }
        else {
                if (rfc == TRUE) {
                    tp.tp_id = TPID_RFC1006;
                }
                else {
                        tp.tp_id = TPID_TCP;
                }
          }

        /* create mutexe */
        status = pthread_mutex_init(&cond_mutex, NULL);
    CHECK(status,"mutex_init bad status\n");

        /* create condition variable */
        status = pthread_cond_init(&cond_var, NULL);
    CHECK(status,"cond_init bad status\n");

        /* thread attribute initialisation */
    status = pthread_attr_init(&pthr_attr_std);
    CHECK(status,"attr_init bad status\n");
    status = pthread_attr_setdetachstate(&pthr_attr_std,
                                                    PTHREAD_CREATE_UNDETACHED);
    CHECK(status,"attr_setdetachstate bad status\n");

        /* create the clients threads */
        for (receiver_num = 0; receiver_num < max_thread; receiver_num++) {
                tab_handler[receiver_num].num_th = receiver_num;
                status =
                 pthread_create(&(tab_handler[receiver_num].thread_handler),
                                                &pthr_attr_std,
                    (void *)thread_client,
                                        (void *)&tab_handler[receiver_num].num_th);
                CHECK(status,"pthread_create bad status\n");
        }
        /*
        * set the predicate thread_hold to zero, and broadcast on the
        * condition variable that the 'thread_client' may proceed.
        */
        status = pthread_mutex_lock(&cond_mutex);
        CHECK(status,"mutex_lock bad status\n");
        thread_hold = 0;
        status = pthread_cond_broadcast(&cond_var);
        CHECK(status,"broadcast on cond_var bad status\n");
        status = pthread_mutex_unlock(&cond_mutex);
        CHECK(status,"mutex_unlock bad status\n");

        /* join each of the client threads */
        for (receiver_num = 0; receiver_num < max_thread; receiver_num++) {
                status =
                  pthread_join(tab_handler[receiver_num].thread_handler,
                                        (void **)&pt_status);
                CHECK(status, "pthread_join bad status\n");
                if ((*((int *)pt_status) == receiver_num) && (trace))
                        fprintf(stdout, "thread %d terminated normally\n",

receiver_num);
                status =
                 pthread_detach(tab_handler[receiver_num].thread_handler);
                CHECK(status,"pthread_detach bad status\n");
        }
```

```
        status = pthread_mutex_destroy(&cond_mutex);
        CHECK(status,"pthread_mutex_destroy bad status\n");
        status = pthread_cond_destroy(&cond_var);
        CHECK(status,"pthread_cond_destroy bad status\n");

        pthread_exit((void *) NULL);
}
```

The Server

```
/*
 *@BULL_COPYRIGHT@
*/
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stropts.h>
#include <fcntl.h>
#include <signal.h>
#include <xti3to4.h>
#include <xti.h>
#include <xti_ns.h>

#define LSIZE                 20
#define INFILE_DFLT "infile"
#define SERVERNAME_DFLT       "cots_server"
#define LBUF_DFLT   1024
#define LBUF_MAX    8192
#define DISCONNECT  -1
#define MAX_THREAD     100

char      *infile = INFILE_DFLT;
char      *servername = SERVERNAME_DFLT;
int       iso = FALSE;
int     rfc = FALSE;
int       lbuf = LBUF_DFLT;
int     trace = FALSE;
int       sender_num;
int       status;
struct th_hand {
      pthread_t thread_handler;
      int num_th;
} ;
struct th_hand tab_handler[MAX_THREAD];
int       listen_fd;
struct    xtitp tp;
struct t_bind *bindret;
int conn_fd[MAX_THREAD];

extern char *optarg;
extern int getopt();

void connrelease(my_number)   /* -- REL : CONNECTION RELEASE -- */
int my_number;
{
int evt;

          /* until release or disconnection indication arrives */
          while ((evt = t_look(conn_fd[my_number])) == 0) {
                    sleep(1);
          }
          if (evt < 0) {
                    fprintf(stderr,
                         "server %d : t_look failed for conn_fd, error : %s\n",
                         my_number, t_strerror(t_errno));
        }
        else {
          switch (evt) {
```

```
          case T_DISCONNECT:
                    if (t_rcvdis(conn_fd[my_number], NULL) < 0 ){
                              fprintf(stderr,
                                "server %d : t_rcvdis failed, error : %s\n",
                                my_number, t_strerror(t_errno));
                        break;
                    }
                    fprintf(stdout,
                            "server %d : abortive disconnection received ok\n",
                        my_number);
                    break;

          case  T_ORDREL:
                    if (t_rcvrel(conn_fd[my_number]) < 0 ){
                              fprintf(stderr,
                                "server %d : t_rcvrel failed, error : %s\n",
                                my_number, t_strerror(t_errno));
                        break;
                    }
                    fprintf(stdout, "server %d : orderly release received\n",
                        my_number);
                    if (t_sndrel(conn_fd[my_number]) < 0 ){
                              fprintf(stderr,
                                "server %d : t_sndrel failed, error : %s\n",
                                my_number, t_strerror(t_errno));
                        break;
                    }
                    fprintf(stdout,
                            "server %d : orderly release acknowledged ok\n",
                        my_number);
                    break;

          default:
                    fprintf(stderr, "server %d : error evt: %x\n", my_number, evt);
                    break;
          }
          }
}
```

send_message(void *arg)

```
                    /* -- XFER : DATA TRANSFER -- */
{
        int nbytes;
        FILE *infp;
        char buf[LBUF_MAX];
        char lgth_file[LSIZE];
```

int my_number = *((int *)arg);

```
        if (trace) {
                    fprintf(stdout, "server %d : thread server input \n",
                        my_number);
        }
```

if (my_number < 0) {
fprintf(stdout,"server %d : my_number not in range\n",
my_number);
pthread_exit(NULL);

```
        }
        fflush(stdout);
        if ((infp = fopen(infile, "r")) == NULL) {
                perror("cannot open input file");
```

pthread_exit(NULL);

```
        }
        sprintf(lgth_file,"%d",filesize(infp));
        fprintf(stdout,
                "server %d : %d bytes in input file \"%s\" to send\n",
                my_number, filesize(infp), infile);
        if (t_snd(conn_fd[my_number], lgth_file, LSIZE, 0) < 0) {
```

```
                if (t_errno == TLOOK) {
                        fprintf(stderr,
                          "server %d : t_snd : connection aborted\n",
                          my_number);
                        pthread_exit(NULL);
                }
                    fprintf(stderr,
                        "server %d : t_snd failed for conn_fd, error : %s\n",
                        my_number, t_strerror(t_errno));
                    pthread_exit(NULL);
        }
        while ((nbytes = fread(buf, 1 , lbuf, infp)) > 0) {
                if (t_snd(conn_fd[my_number], buf, nbytes, 0) < 0) {
                        if (t_errno == TLOOK) {
                                    fprintf(stderr,
                                "server %d : t_snd : connection aborted\n",
                                my_number);
                                    pthread_exit(NULL);
                        }
                              fprintf(stderr,
                          "server %d : t_snd failed for conn_fd, error : %s\n",
                           my_number, t_strerror(t_errno));
                              pthread_exit(NULL);
                }
       }
          if (fclose(infp) < 0)
                    fprintf(stderr, "server %d : fclose failed\n", my_number);
       /* wait  release or disconnection indication */
        connrelease(my_number);
          t_close(conn_fd[my_number]);
        if (trace) {
                    fprintf(stdout, "server %d : thread output \n" ,my_number);
        }
          pthread_exit(NULL);
}

int filesize(fd)
FILE *fd;
{
struct stat file_state;

          if (fstat(fileno(fd), &file_state)) return -1;
          return file_state.st_size;
}

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

        if ((resfd = t_open(tp.tp_name, O_RDWR, NULL)) < 0) {
                t_error("t_open for responding fd failed");
                pthread_exit((void *)NULL);
        }

        if ( iso == TRUE ) {
                bindret->qlen = 0;
                if (t_bind(resfd, bindret, NULL) < 0) {
                        t_error("t_bind for responding fd failed");
                              pthread_exit((void *)NULL);
                }
        }
        /* for TCP the address is not necessary */
        else {
                 if (t_bind(resfd, NULL, NULL) < 0) {
```

```
                        t_error("t_bind for responding fd failed");
                                pthread_exit((void *)NULL);
                }
        }
        /*call->sequence is actually initialized */
        /*because returned by t_listen*/
        call->addr.len = 0;
        call->opt.len = 0;
        call->udata.len = 0;
        if (t_accept(listen_fd, resfd, call) < 0) {
                if (t_errno == TLOOK) {
                        if (t_rcvdis(listen_fd , NULL) < 0) {
                                t_error("t_rcvdis failed for listen_fd");
                                        pthread_exit((void *)NULL);
                        }
                        if (t_close(resfd) < 0) {
                                t_error("t_close failed for responding fd");
                                        pthread_exit((void *)NULL);
                        }
                        return(DISCONNECT);
                }
                t_error("t_accept failed");
                    pthread_exit((void *)NULL);
        }
        return(resfd);
}

usage(type, proc, value)
int type;
char *proc;
char *value;
{
        switch (type) {
                case 1:
                        fprintf(stderr, "error on length of buffer %s\n",
                                        value);
                        break;
                default:
                        break;
        }
        fprintf(stderr,
                "usage: %s [-bB] [-fF] [-i|r] [-sS] [-v]\n", proc);
        fprintf(stderr,
                "   -bB       set length of sending buffer to B ");
        fprintf(stderr,"(default is %d, max is %d)\n", LBUF_DFLT, LBUF_MAX);
        fprintf(stderr,
                "   -fF       set input file name to F ");
        fprintf(stderr,"(default is \"%s\")\n", INFILE_DFLT);
        fprintf(stderr,
                "   -i        set transport provider to ISO ");
        fprintf(stderr,
                "or -r        set transport provider to RFC1006 ");
        fprintf(stderr,"(default is TCP)\n");
        fprintf(stderr,
                "   -sS       set server name to S ");
        fprintf(stderr,"(default is \"%s\")\n", SERVERNAME_DFLT);
        fprintf(stderr,
                "   -v        set on verbose mode (default is OFF)\n");
}
```

```
/*************************************************************/
/* COTS_SERVER : [-blbuf] [-finfile] [-i] [-sservername] [-v] */
/*************************************************************/

main(argc,argv)
int argc;
char *argv[];
{
#define LBUF 250

        int c;
        struct t_bind *bind;
        struct t_call *call;
        int i;
        char *pt;
        char bufopt[LBUF];
        char bufaddr[LBUF];
        struct t_info info;

        while ((c = getopt(argc,argv,"b:f:irs:v")) != -1) {
                switch (c) {
                case 'b':
                        lbuf = atoi(optarg);
                        if (lbuf < 1 || lbuf > LBUF_MAX) {
                                usage(1, argv[0], optarg);
                                        pthread_exit((void *)NULL);
                        }
                        break;
                case 'f':
                        infile = optarg;
                        break;
                case 'i':
                        if (rfc==FALSE)
                                iso = TRUE;
                        else {
                                usage(-1, argv[0], NULL);
                                        pthread_exit((void *)NULL);
                        }
                        break;
                case 'r':
                        if (iso == FALSE)
                                rfc = TRUE;
                        else {
                                usage(-1, argv[0], NULL);
                                        pthread_exit((void *)NULL);
                        }
                        break;
                case 's':
                        servername = optarg;
                        break;
                case 'v':
                        trace = TRUE;
                        break;
                default :
                        usage(-1, argv[0], NULL);
                              pthread_exit((void *)NULL);
                }
        }

        if (iso == TRUE) {
                tp.tp_id = TPID_OSI_COTS;
        }
        else {
                if (rfc == TRUE) {
                        tp.tp_id = TPID_RFC1006;
                }
```

```
        else {
                tp.tp_id = TPID_TCP;
        }
}

  if (t_gettp(&tp) < 0) {
            if (t_error_ns("server : t_gettp failed") < 0)
                     fprintf(stderr,
                     "server : t_error_ns failed\n");
            pthread_exit((void *)NULL);
}
if ((listen_fd = t_open(tp.tp_name, O_RDWR, NULL)) < 0) {
        t_error("t_open failed for listen_fd");
            pthread_exit((void *)NULL);
}
if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND_STR, T_ALL))
          == NULL) {
        t_error("t_alloc failed for bind");
            pthread_exit((void *)NULL);
}
if (t_getladdr(&tp, servername, &bind->addr) < 0) {
            if (t_error_ns("server : t_getladdr failed") < 0)
                     fprintf(stderr,
                     "server : t_error_ns failed\n");
            pthread_exit((void *)NULL);
}

/* the server endpoint is used to listen for connect indication */
bind->qlen = 50;
if ((bindret = (struct t_bind *)t_alloc(listen_fd, T_BIND_STR, T_ALL))
             == NULL) {
        t_error("t_alloc failed for bindret");
            pthread_exit((void *)NULL);
}
if (t_bind(listen_fd, bind, bindret) < 0) {
        t_error("t_bind failed for listen_fd");
            pthread_exit((void *)NULL);
}
  if (trace) {
        fprintf(stdout,"server : ");
        for (i=0, pt=bindret->addr.buf; i<bindret->addr.len; i++, pt++)
                fprintf(stdout, "0x%x", *pt);
            fprintf(stdout,"\n(qlen=%d)\n",bindret->qlen);
}

/* -- ESTB1 : CONNECTION ESTABLISHMENT first phase -- */

if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL_STR, T_ALL))
                  == NULL)
{
        t_error("t_alloc failed for call");
            pthread_exit((void *)NULL);
}
if (info.addr == -1)
/* -1  : no limit       ==> field not alocated by t_alloc */
{
        call->addr.buf          = bufaddr;
        call->addr.maxlen       = LBUF;
}
if (info.options == -1)
/* -1  : no limit       ==> field not alocated by t_alloc */
{
        call->opt.buf           = bufopt;
        call->opt.maxlen        = LBUF;
}
sender_num = 1;
```

```
        while (1) {
                if (trace) {
                        fprintf(stdout, "server is waitting on t_listen\n");
                }
                while (t_listen(listen_fd, call) < 0) {
                        if (t_errno != TNODATA) {
                                t_error("t_listen failed for listen_fd");
                                        pthread_exit((void *)NULL);
                        }
                } /* active wait */
                if ((conn_fd[sender_num] = accept_call(listen_fd, call))
                                                        != DISCONNECT) {
        tab_handler[sender_num].num_th = sender_num;
        status = pthread_create(
                                        &tab_handler[sender_num].thread_handler,
            &pthread_attr_default,
            (void *)send_message,
            (void *)&tab_handler[sender_num].num_th);
                        if (status == -1)
            perror ("pthread_create bad status\n");

                        if (sender_num == 99)
                        sender_num = 0;
              sender_num ++;
                }
        /* t_free of bind and call structures be effectively */
        /* realised when the server process will be killed */
        /* (t_alloc use the general memory allocation function) */
          }

          pthread_exit((void *)NULL);

        /* -- END ESTB1 --*/

} /* end main server */
```

Appendix A. XTI Test Tools

Three test tools are provided with Bull-enhanced XTI:

- **bench** Tool, on page A-2, tests the Bull-enhanced XTI performance through one or several connections,
- **tconnect** Tool, on page A-8, checks that a user-defined number of parallel connections is acceptable over a Transport Provider,
- **xtistat** Tool, on page A-13, provides global statistics of the XTI activity or the XTI activity of each Transport Endpoint.

bench Tool

The **bench** tool calculates the throughput and the number of messages sent per second over a Transport Provider through one connection or several connections in parallel.

The **bench** tool is composed of two components:

- the **benchd** daemon which simulates a server-application, on page A-3,
- the **bench** command which simulates a client-application, on page A-5.

benchd must be run before **bench**.

The client **bench** sends messages to the server **benchd**. In interactive mode, these messages are sent back by the server **benchd** to the client **bench**.

Several clients (**bench**) can communicate with the server (**benchd**) at the same time or sequentially:

- the main process of the client forks for every connection and the child manages this connection. So, the number of client processes is C+1 if C is the number of connections requests (in connectionless mode, the main process manages the data transfer). All the connections are opened before any data transfer occurs.
- the main process of the server receives connections indications from the client, it forks for every accepted connection and the child manages this connection. So, the number of server processes is C+1 if C is the number of connections indications received (in connectionless mode, the main process manages the data transfer).

bench can stop by itself after having sent all its messages or can be killed by the user.

benchd must be killed by the user at the end of the test.

The throughput and the number of messages sent per second are reported periodically and at the end of the test.

The **bench** tool tests the Bull-enhanced XTI performance using the XTI library over:

- OSI Connection-Oriented,
- OSI ConnectionLess,
- TCP/IP,
- UDP/IP,
- NetShare (RFC 1006),
- X.25.

It can test as well TCP/IP or UDP/IP using the sockets.

benchd Daemon

Purpose

Calculate the throughput and the number of messages sent per second over a Transport Protocol through one connection or several connections in parallel.

(Server-component of the **bench** tool).

Syntax

benchd [**–x**] [**–t** | **–u** | **–o** | **–ol** | **–r** | **–x25**] [**–i**] [**–e** *E* | **–p** *P* | **–T** *t*] [**–m***M*]

Description

At first, **benchd** prints on the standard output:

- the Transport Provider,
- the use of the interactive mode (eventually),
- the local address of the Server and the Server name,
- the trace period.

Then for each connection:

- a line report displays periodically (see the –**m***M* option) the throughput in bytes and number of messages received by **benchd** per second.
- a line report displays at the end of the connection the minimum, average and maximum throughput in bytes and the average of number of messages received per second.

benchd reports on the standard error, all errors (syntax errors in the command line, initialization errors, data exchange errors, ...).

Flags

–x Tests the Bull-enhanced XTI performance using the XTI library.
By default, tests TCP/IP using the sockets.

–t|–u|–o|–ol|–r|–x25
Sets the Transport Provider:
–t means TCP
–u means UDP
–o means OSI_COTP
–ol means OSI_CLTP
–r means NetShare (RFC 1006)
–x25 means X.25
The default Transport Provider is TCP.
This parameter must be the same for **benchd** and **bench** (except for **–u** option which must be used by **benchd** if **–b** option is used by **bench).**

–i Sets the interactive mode. In this mode all the messages received by **benchd** are sent back to **bench**.
This option is valid only for a connection-oriented Transport Provider (TCP, OSI_COTP, NetShare (RFC 1006) and X.25).

–e *E* Sets the Server name. The server name must be present in the services data base (/etc/services or /etc/xtiservices).
By default, the server is **benchd**.
The **–p**, **–e** and **–T** parameters are mutually exclusive.

–p *P* Sets the Port number. This parameter is valid for test of TCP or UDP, if the user does not want to use the name server.
The **–p**, **–e** and **–T** parameters are mutually exclusive.

–T *t* Sets the TSEL if the Transport Provider is OSI_COTP (**–o**) or NetShare (RFC 1006) (**–r**)
Sets the SAI if it is X.25 (**–x25**)

–m *M* Sets the delay in seconds between throughput reports displays.
Each connection uses this delay to trace a throughput report. So each *M* seconds, as many lines reported as number of existing connections are displayed.
The default value is 10 seconds.

Examples

1. To test Bull-enhanced XTI performance using OSI_COTP Transport Provider and the **benchd** server:

   ```
   benchd -x -o
   ```

2. To test TCP/IP using socket interface with the port 9999:

   ```
   benchd -p9999
   ```

3. To test XX25 performance:

   ```
   benchd -x -x25
   ```

4. To test Bull-enhanced XTI performance using OSI_CLTP Transport Provider and the **benchd** server:

   ```
   benchd -x -ol
   ```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/etc/xtihosts
/etc/xtiservices
/etc/hosts
/etc/services

Suggested Reading

Prerequisite Information

bench Tool

Related Information

bench Command

bench Command

Purpose

Calculate the throughput and the number of messages sent per second over a Transport Protocol through one connection or several connections in parallel.

(Client-component of the **bench** tool).

Syntax

bench [**–x**] [**–t** | **–u** | **–o** | **–ol** | **–r** | **–x25** | **–b**] [**–i**] [**–h** *H*] [**–e** *E* | **–p** *P* | **–T** *t*] [**–n** *N*] [**–l** *L*] [**–s** *S*] [**–k** *K*] [**–c** *C*] [**–d** *D*] [**–m** *M*]

Description

At first, **bench** prints on the standard output:

- the Transport Provider,
- the use of the interactive mode (eventually),
- the Host, the remote address of the Server and the Server name (or the Port number),
- the length and the number of the messages to be sent, the TPDU size,
- the number of connections to be opened and the delay between connections requests,
- the trace period.

On each opened connection **bench** sends messages to the server **benchd**. For each connection:

- a line report displays periodically (see the **–m***M* option) the throughput in bytes and number of messages sent by **bench** (and received in the interactive mode) per second.
- a line report displays at the end of the connection (last message sent or break from user) the minimum, average and maximum throughput in bytes and the average of messages sent (and received in the interactive mode) per second.

bench reports on the standard error, all errors (syntax errors in the command line, initialization errors, data exchange errors, invalid tpdu size, invalid transport size ...).

Flags

–x Tests the Bull-enhanced XTI performance using the XTI library.
By default, tests TCP/IP using the sockets.

–t|–u|–o|–ol|–r|–x25
Sets the Transport Provider:
–t means TCP,
–u means UDP,
–o means OSI_COTP,
–ol means OSI_CLTP,
–r means NetShare (RFC 1006)
–x25 means X.25
–b means UDP in broadcast mode (this option may be used only for test with socket interface).
The default Transport Provider is TCP.
This parameter must be the same for **bench** and **benchd**, except for **–b** option which may be used with **–u** option on **benchd**.

–i Sets the interactive mode. In this mode all the messages sent by **bench** are sent back by **benchd**.
This option is valid only for a connection-oriented Transport Provider (TCP, OSI_COTP, NetShare (RFC 1006) and X.25.

–h *H* Sets the server Host name. The server host must be present in the host data base (/etc/hosts or /etc/xtihosts).
By default, the server host name is “localhost”.

–e*E* Sets the Server name. The server name must be present in the services data base (/etc/services or /etc/xtiservices).
By default, the server is **benchd**.
The **–p**, **–e** and **–T** parameters are mutually exclusive.

–p*P* Sets the Port number. This parameter is valid for test of TCP or UDP, if the user does not want to use the name server.
The **–p**, **–e** and **–T** parameters are mutually exclusive.

–T *t* Sets the TSEL if the Transport Provider is OSI_COTP (**–o**), OSI_CLTP (**–ol**) or NetShare (RFC 1006) (**–r**).
Sets the SAI if it is X.25 (**–x25**)

–n*N* Sets the number of messages to be sent.
The default value is infinite. In this case, the user must kill **bench** to stop it.

–l*L* Sets the length of the messages to be sent.
For CLTS, the length must be less than the Network Provider NSDU (no segmentation provided on CLTS).
The default length is 4096 bytes for all Transport Providers, except for NetShare (RFC 1006) whose default length is 4000.

–s*S* Sets the size of TPDUs. This parameter is valid for test of Bull-enhanced XTI over the Transport Provider OSI_COTP or NetShare (RFC 1006)
The default TPDU size is 4096 bytes.

–k*K* Sets the transport class. This parameter is valid for test of Bull-enhanced XTI over the Transport Provider OSI_COTP.
K may be equal to class 0, 2, 3 or 4.

–c*C* Sets the number of connections to be opened. This option is valid with connection-oriented Transport Provider only (TCP, OSI_COTP and NetShare (RFC 1006).
The default number of connections to open is 1.

–d*D* Sets the delay in seconds between connections requests.
The default value is no delay.

–m *M* Sets the delay in seconds between throughput reports displays.
Each connection uses this delay to trace a throughput report. So each *M* seconds, as many lines reported as number of existing connections are displayed.
The default value is 10 seconds.

Examples

1. To test Bull-enhanced XTI performance using OSI_COTP Transport Provider, the **benchd** server on the “host_server” and sending messages of 128 bytes:

   ```
   bench -x -o -l128 -hhost_server
   ```

2. To test TCP/IP using socket interface with the port 9999 on the “localhost” and sending 500 messages of 4096 bytes:

   ```
   bench -p9999 -n500
   ```

3. To test XX25 performance sending messages of 1024 bytes:

```
bench -x -x25 -l1024
```

4. To test Bull-enhanced XTI performance using OSI_CLTP Transport Provider, the **benchd** server on the “host_server” and sending messages of 1024 bytes :

```
bench -x -ol -l1024 -hhost_server
```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/etc/xtihosts
/etc/xtiservices
/etc/hosts
/etc/services

Suggested Reading

Prerequisite Information

bench Tool

Related Information

benchd Daemon

tconnect Tool

The **tconnect** tool checks that a user-defined number of parallel connections is acceptable over a Transport Provider. The connections may be opened connections (without data exchange) or active connections (with data exchange).

The **tconnect** tool is composed of two components:

- the **tconnectd** daemon which simulates a server-application, on page A-9,
- the **tconnect** command which simulates a client-application, on page A-11.

tconnectd must be run before **tconnect**.

First, the two components try to open all the connections and generate reports if an open error occurs. Then a data exchange is established on each connection, if required.

Several clients can exchange with the server:

- the main process of the client forks and every child manages F connections. So, the number of client processes is 1+C/F (+1 if C modulo F is not 0) if C is the number of connections requests and F the number of connections per process.
- the main server process forks and the child receives connections indications from the client. When it cannot accept any more connection, it sends a signal to its father which forks again to manage the following connections. So, the number of server processes is 1+C/N (+1 if C modulo N is not 0) if C is the number of connections indications and N the number of connections allowed per process.

tconnect can stop by itself after having sent all its messages or can be killed by the user.

tconnectd must be killed by the user at the end of the test.

The **tconnect** tool tests the Bull-enhanced XTI performance using the XTI library over:

- OSI Connection-Oriented,
- TCP/IP,
- NetShare (RFC 1006),
- X.25.

It can test as well TCP/IP or UDP/IP using directly the sockets.

CAUTION:
To determine the maximum number of connections over a Transport Provider, do not forget to take in account that other transport applications may also have opened connections.
This tool is not dedicated to CLTS users.

tconnectd Daemon

Purpose

Check that a user-defined number of parallel connections is acceptable over a Transport Provider. The connections may be opened connections (without data exchange) or active connections (with data exchange).

(Client-component of the **tconnect** tool).

Syntax

tconnectd [**–x**] [**–t** | **–o** | **–r** | **–x25**] [**–e***E* | **–p***P*]

Description

tconnectd prints on the standard output:

- the Transport Provider,
- the Server name or the Port number.

tconnectd reports on the standard error, all errors (syntax errors in the command line, initialization errors, data exchange errors, ...).

Flags

–x	Tests the Bull-enhanced XTI performance using the XTI library. By default, tests TCP/IP using the sockets.
–t|–o|–r|–x25	Sets the Transport Provider: **–t** means TCP, **–o** OSI_COTP, **–r** NetShare (RFC 1006) and **–x25** X.25. The default Transport Provider is TCP. This parameter must be the same for **tconnectd** and **tconnect**.
–e*E*	Sets the Server name. The server name must be present in the services data base (/etc/services or /etc/xtiservices). By default, the server is **tconnectd**. The **–p** and **–e** parameters are mutually exclusive.
–p*P*	Sets the Port number. This parameter is valid for test of TCP or UDP using directly the sockets, if the user does not want to use the name server. The **–p** and **–e** parameters are mutually exclusive.

Examples

1. To test Bull-enhanced XTI performance using OSI_COTP Transport Provider and the **tconnectd** server:

   ```
   tconnectd -x -o
   ```

2. To test TCP/IP using directly socket interface with the port 9999:

   ```
   tconnectd -p9999
   ```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/etc/xtihosts
/etc/xtiservices
/etc/hosts
/etc/services

Suggested Reading

Prerequisite Information

tconnect Tool

Related Information

tconnect Command

tconnect Command

Purpose

Check that a user-defined number of parallel connections is acceptable over a Transport Provider. The connections may be opened connections (without data exchange) or active connections (with data exchange).

(Client-component of the **tconnect** tool).

Syntax

tconnect [**–x**] [**–t** | **–o** | **–r** | **–x25**] [**–h***H*] [**–e***E* | **–p***P*]
[**–n***N*] [**–l***L*] [**–c***C* [,*F*]] [**–d***D*]

Description

tconnect prints on the standard output:

- the Transport Provider,
- the Host, the remote address of the Server and the Server name (or the Port number),
- the length and the number of the messages to be sent,
- the total number of connections to be opened, the number of connections to be opened per process and the delay between connections requests.

tconnect reports on the standard error all errors (syntax errors in the command line, initialization errors, data exchange errors, ...).

Flags

–x Tests the Bull-enhanced XTI performance using the XTI library.
By default, tests TCP/IP using the sockets.

–t|–o|–r|–x25 Sets the Transport Provider:
–t means TCP
–o means OSI_COTP
–r means NetShare (RFC 1006)
–x25 means X.25
The default Transport Provider is TCP.
This parameter must be the same for **tconnect** and **tconnectd**.

–h *H* Sets the server Host name. The server host must be present in the host data base (/etc/hosts or /etc/xtihosts).
By default, the server host name is “localhost”.

–e*E* Sets the Server name. The server name must be present in the services data base (/etc/services or /etc/xtiservices).
By default, the server is **tconnectd**.
The **–p** and **–e** parameters are exclusive.

–p*P* Sets the Port number. This parameter is valid for test of TCP using directly the sockets, if the user does not want to use the name server.
The **–p** and **–e** parameters are exclusive.

–c*C*,*F* Sets the total number of connections to open to *C* and the number of connections per forked process to *F*.
The default values are 1 for *C* and 32 for *F*.

–d*D* Sets the delay in seconds between connections requests.
The default value is no delay.

–n*N* Sets the number of messages to be sent.
The default value is 128.

–l*L* Sets the length of the messages to be sent.
The default length is 1024 bytes.

Examples

1. To check whether one connection over OSI_COTP Transport Provider may be used to communicate with the **tconnectd** server on the “localhost” without sending any data:

   ```
   tconnect –x –o –n0
   ```

2. To check whether 256 connections (each process managing 64 connections) over TCP/IP using directly socket interface with the port 9999 may be used to communicate with the **tconnectd** server on the “localhost” and to send 128 messages of 512 bytes:

   ```
   tconnect –p9999 –c256,64 –l512
   ```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

/etc/xtihosts
/etc/xtiservices
/etc/hosts
/etc/services

Suggested Reading

Prerequisite Information

tconnect Tool

Related Information

tconnectd Daemon

xtistat Command

Purpose

Display global statistics of the XTI activity or the XTI activity of each XTI Transport Endpoint.

Syntax

xtistat [**–s**] [**–p** *protocol*] [**–n**] [**–d** [*CTSQEIWplderiaf*]] [**–a**] [**–A**]
[**–i** *sec*] [**–c** *addr*] [**–b**] [**core**]

Flags

–s Display global statistics

–p *protocol* Select a Transport Provider (*osi_cots*, *osi_clts, tcp*, *udp*, *rfc1006, x25*).

–n Disable the interpretation (via the **libxti_ns** library) of protocol addresses.

–d [*CTSQEIWplderiaf*]
Select information to be displayed.
C context address
T trace level
S XTI state
Q queue length
E event
I connect indication count
W high and low water mark
p protocol
l local address
d remote address
e stream errors count
r request count
i indication count
a acknowledge count
f stream flow

–a equivalent to **–d** *Spfld*

–A Display all information

–i *sec* Select continuous display mode. The screen is refreshed each *sec* seconds.

–c *addr* Select the address (in C format) of a defined context block.

–b Dump context block.

core Read in a specified core instead of /dev/kmem.

Examples

1. To display global statistics:

```
xtistat -s

0 Connection Oriented Transport Provider.
0 Connectionless Transport Provider.
1 RFC1006 Transport Provider.
1 TCP Transport Provider.
0 UDP Transport Provider.
0 X.25 Network Provider
0 Other Transport Provider.
```

```
54070 Open.
54068 Close.
27171 Connect requests.
26873 Connect indications.
26873 Connect accepts.
2074208 User send messages.
1875730 TPI send messages.
2124232 User received messages.
1825407 TPI received messages.
0 Memory alloc failures.
0 Bind failures.
54065 Bind success.
27177 Bad state errors.
54040 Error acks.
80906 Ok acks.
```

2. To display TCP connections:

```
xtistat -p tcp
```

```
Protocol M_rcv M_snd Loc Addr ........Rem Addr State
tcp .....8 ....8 ....cots_server/00.0 none ....TS_IDLE
```

3. To display Context, State and Addresses of each connections:

```
xtistat -d CSld
```

```
Ctx blk: 0x5550030
Loc Addr: 00.00.00.00/cots_server
Rem Addr: none
State: TS_IDLE
```

4. To display all information:

```
xtistat -A
```

```
Ctx blk: 0x5550030
Protocol: tcp
FLOW rcv_Q snd_Q Usr Rcv Usr Snd Prov Snd Prov Rcv
 ....0 ....0 ....8 ......8 ......6 .......6
Loc Addr: 00.00.00.00/cots_server
Rem Addr: none
State: TS_IDLE
Trace lev.: 0x38000000
Qlen: 1
Event: none
ConInd cnt: 0
LoWat: 0/0
ERR alloc state tpisz errak opt. bind
 ...0 ....0 ....0 ....0 ....0 ...0
REQ data expdt conrq conrs ordrl ioctl
 ...0....0 ....0 ....0 ....0 ....8
IND data conn. disc. ordrl uderr
 ...0 ...0 ....0 ....0 ....0
ACK bind addr error ok concf
 ...1 ...0 ...0 ....0 .0
```

5. To display Context Dump:

```
xtistat -b -c 0x5b71c2c

Context DUMP:
05b71c2c: 00000000 38000000 00000004 00000001 ....8...........
05b71c3c: 00000000 00000000 00000000 00000000 ................
05b71c4c: 00000000 00000000 00000000 00000000 ................
05b71c5c: 00000000 00000000 00000000 00000000 ................
05b71c6c: 00000000 00000000 00000000 00000000 ................
05b71c7c: 00000000 636f7470 00000000 00000000 ....cotp........
05b71c8c: 00000021 00000001 00000005 78636f74 ...!........xcot
05b71c9c: 73000000 00000002 00000000 00000003 s...............
05b71cac: 00000001 ff000000 00000000 00000000 ................
05b71cbc: 00000000 00000000 00000000 00000000 ................
05b71ccc: 00000000 00000000 00000000 00000000 ................
05b71cdc: 00000000 00000000 00000000 00000000 ................
05b71cec: 00000000 00000000 00000000 00000000 ................
05b71cfc: 00000000 00000000 00000000 00000000 ................
05b71d0c: 00000000 00000000 00000000 00000000 ................
05b71d1c: 00000000 00000000 00000000 00000000 ................
05b71d2c: 00000000 00000000 00000000 00000000 ................
05b71d3c: 00000000 00000000 00000000 00000000 ................
05b71d4c: 00000000 00000000 00000000 00000000 ................
05b71d5c: 00000000 00000000 00000000 00000000 ................
05b71d6c: 00000000 00000000 00000000 00000000 ................
05b71d7c: 00000000 00000000 00000000 00000000 ................
05b71d8c: 00000000 00000000 00000000 00000000 ................
05b71d9c: 00000000 00000000 00000000 00000000 ................
05b71dac: 00000000 00000000 00000000 00000000 ................
05b71dbc: 00000000 00000000 00000000 00000000 ................
05b71dcc: 00000000 00000000 00000000 00000000 ................
05b71ddc: 00000000 00000000 00000008 00000000 ................
05b71dec: 00000000 00000001 00000001 00000000 ................
05b71dfc: 00000000 00000000 00000000 00000004 ................
05b71e0c: 00000000 00000000 00000000 00000000 ................
05b71e1c: 00000008 00000008 00000006 00000006 ................
05b71e2c: 00000000 00000000 00000001 05b74418 ..............D.
05b71e3c: 05b7408c 05b74500 05be002c 00000000 ..@...E....,....
```

Implementation Specifics

This command is part of **xti_api** software, but is not part of the XTI standard.

Files

Suggested Reading

Related Information

Appendix B. File Formats

This appendix provides description of the XTI Data Base files:

xtihosts File

Purpose

XTI OSI and **XX25 Hosts** Data Base

- An **OSI Host** object defines a path within the transport to access a remote host.
- An **XX25 Host** object defines a path within the network to access a remote host.

Description

For each host a single line should be present with the following format:

```
<remote_addr> <local_addr> <lsap> <network_type> <name> <aliases>
```

where:

`<remote_addr>`

Address used to access the remote transport. Depending on `<network_type>`

- For **XTI OSI**

Network type	Address Description
0x0101	SNPA = **X.121 address** (15 decimal digits max)
0x0102	not significant = **NULL**
0x0103	SNPA = **MAC address** (6 bytes max) For example: Ethernet, Token Ring MAC address or FDDI address
0x0306 0x0506	NSAP = **Network Service Access Point** (20 bytes max)

- For **XX25**

Virtual Circuit type	Address Description
0x2501	SVC = **X.121 address** (15 decimal digits max)
0x2502	not significant = **NULL**

`<local_addr>`

Address through which the connection goes out to the remote host. Depending on `<network_type>`

- For **XTI OSI**

Network type	Address Description
0x0101	SNPA = **X.121 address** (15 decimal digits max)
0x0102	local PVC name (8 bytes max)
0x0103	SNPA = **MAC address** (6 bytes max) For example: Ethernet, Token Ring MAC address or FDDI address
0x0306 0x0506	NSAP = **Network Service Access Point** (20 bytes max)

- For **XX25**

Virtual Circuit type	Address Description
0x2501	SVC = **X.121 address** (15 decimal digits max)
0x2502	local PVC name (8 bytes max)

`<lsap>` 0x20 for **DSA**
0xfe for **ISO**
`lsap` is not significant for **XX25**

`<network_type>`
Numeric identifier of the network used to communicate.

- For **XTI OSI**

Network type	Network Description
0x0101	**CONS**/**WAN**/**SVC** = COTS over CONS on S VC For example: X25
0x0102	**CONS**/**WAN**/**PVC** = COTS over CONS on PVC For example: X25
0x0103 LSAP=20	**I_CLNS**/**LAN** = COTS over inactive CLNS (Non–full OSI conformance) with Data Link Service Access Point = **DSA** For example Ethernet, Token Ring, FDDI
0x0103 LSAP=fe	**I_CLNS**/**LAN** = COTS over inactive CLNS (Full OSI conformance) with Data Link Service Access Point = **OSI**
0x0306	**CLNS** = COTS and CLNS over LAN and WAN (Full OSI conformance)
0x0506	**SPEE** = COTS over CLNS (on LAN) or COTS over CONS (on WAN)

- For **XX25**

Virtual Circuit type	Description
0x2501	**X.25 SVC** = Switched Virtual Circuit
0x2502	**X.25 PVC** = Permanent Virtual Circuit

`<name>` Host name,
(character string containing no more than 40 characters)

`<aliases>` Aliases for Host name,
(maximum two aliases separated by a blank. Each alias is a character string containing no more than 40 characters)

Items are separated by any number of blanks and/or tab characters.

Host names may contain any printable character other than a field delimiter, newline or comment character.

Example

```
# /etc/xtihosts:    xti hosts database
#############################################################
#
#Remote Host Addrss Local Host Address  Lsap  Netser    Host name
#                                                      and aliases
0x02                NULL            0xfe  0x0306    localhost/tpid_osi_cots
138000000000002     138000000000001 0xfe  0x0101    h_x25_svc/tpid_osi_cots
                                                          old_nt_1 ex_1
0x003001020304      0x003001020304  0x20  0x0103    h_nullip_dsa/tpid_osi_cots
                                                          old_nt_2 ex_2
0x0a0b0c0d          NULL            0xfe  0x0306    h_fullip/tpid_osi_cots
                                                          old_nt_3 ex_3
0x00300102030b      0x00300102030a  0xfe  0x0103    h_nullip_osi/tpid_osi_cots
                                                          old_nt_4 ex_4
0xabef              NULL            0xfe  0x0506    h_spee/tpid_osi_cots
                                                          old_nt_5 ex_5
NULL                ”PVCone”        0xfe  0x0102    h_x25_pvc/tpid_osi_cots
                                                          new_type ex_6
138002              138001          0xfe  0x2501    h_xx25_svc/npid_x25_cons
NULL                138003          0xfe  0x2502    h_xx25_pvc/npid_x25_cons
0x12                NULL            0xfe  0x0306    localhost/tpid_osi_clts
```

Implementation Specifics

This file is used by XTI Name Server, part of **xti_api** software.

Files

/etc/xtihosts

Suggested Reading

Related Information

How to Manage XTI OSI Hosts, on page 3-11.

xtihost Command.

xtiprotocols File

Purpose

XTI and **XX25 Protocols** Data Base.

Description

The **xtiprotocols** file lists the XTI Transport Providers and XX25 Network Provider implemented in Bull-enhanced XTI in the following format:

<provider_name> <provider_id> <device_name> <service_type> <provider_alias>

where:

<provider_name>
Name of the transport provider (for instance, *tpid_osi_cots, tpid_osi_clts, tpid_tcp, tpid_osi_rfc1006*, *npid_x25_cons*).

<provider_id>
Provider identifier as defined in **xti_api/xti_ns.h** include file**.**

<device_name>
Name of the special file to be used to get a transport endpoint opened on the transport provider.
The value '`none`' indicates that no access is currently allowed on this transport provider.

<service_type>
is the type of service offered:
cots, connection-oriented transport protocol
cots_ord, connection-oriented transport protocol with orderly release
clts, connection-less transport protocol
cons, connection-oriented network protocol.

<provider_alias>
List of maximum two alternative names for the transport provider.

Items are separated by any number of blanks and/or tab characters.

Transport provider names may contain any printable character other than a field delimiter, newline or comment character.

Examples

```
# XTI Transport Provider (TP) file
# ################################
#
# Name          Id.     Device          Type       Alias
#
# Do not remove or change the following lines
tpid_osi_cots   0       /dev/xti/cotp   cots       TPID_OSI_COTS
tpid_osi_clts   1       /dev/xti/cltp   clts       TPID_OSI_CLTS
tpid_tcp        2       /dev/xti/tcp    cots_ord   TPID_TCP
tpid_udp        3       /dev/xti/udp    clts       TPID_UDP
tpid_rfc1006    4       /dev/xti/tp1006 cots       TPID_RFC1006
npid_x25_cons   5       /dev/dat/xpi_xd cons       NPID_X25_CONS
```

Implementation Specifics

This file is used by XTI Name Server, part of **xti_api** software.

Files

/etc/xtiprotocols

Suggested Reading

Related Information

xtiservices File.

xtiservices File

Purpose

XTI and **XX25 Services** Data Base.

Description

The **xtiservices** file lists the services available in the network and identified:

by a **TSEL** (Transport SELector) for an **XTI** service,

by an **SAI** (Subsequent Application Identifier) for an **XX25** service.

For each service a single line is present with the following format:

```
<name> <tsel-sai>/<provider_name> <aliases>
```

`<name>` the name of the defined service,
(character string containing no more than 40 characters).

`<tsel-sai>` the OSI Transport SELector or XX25 Subsequent Application Identifier associated to the defined service.
The maximum length of the value is 32 bytes and the format is one of the following :

Format	Syntax description for TSEL–SAI
a list of numerals	separated by commas, for initialization (for example 9,8,7). Each digit for the attribute in a database is set to a digit with the corresponding decimal value (9,8,7 is 0x09,0x08,0x07)
a initializing string	(for example 13829) the resulting value of the attribute in a database is the ascii value of the string (13829 is 0x31,0x33,0x38,0x32,0x39),
a list of hexadecimals	separated by commas, for initialization (for example 0xab,0x02).

`<provider_name>`
the provider name as defined in the **XTI Protocols** Data Base:
/etc/xtiprotocols (string max 20).

`<aliases>` the aliases for name
(maximum two aliases separated by a blank; each alias is a character string containing no more than 40 characters).

Items are separated by any number of blanks and/or tab characters.

Service names may contain any printable character other than a field delimiter, newline or comment character.

Note: As NetShare (RFC 1006) is equivalent to an OSI Class 0 Connection-oriented Transport Provider, the provider name to be specified is `tpid_osi_cots` for any service accessed by NetShare (RFC 1006).

Example

```
#
# /etc/xtiservices:     xti services database
#
# XTI services file
################################################################
#
# Service       TSEL-SAI/Protocol               Aliases
#
cots_server     0x01020301/tpid_osi_cots        COTS_SERVER
                                                COTS_server

select_server   0x01020302/tpid_osi_cots        SELECT_SERVER
                                                SELECT_server

poll_server     0x01020303/tpid_osi_cots        POLL_SERVER
                                                POLL_server

benchd          0x01020304/tpid_osi_cots        BENCHD xtibenchd

tconnectd       0x01020305/tpid_osi_cots        TCONNECTD
                                                xtitconnectd

cons_server     0xC4020301/npid_x25_cons        CONS_SERVER
                                                CONS_server
benchd          0xC4020304/npid_x25_cons        BENCHD xtibenchd
tconnectd       0xC4020305/npid_x25_cons        TCONNECTD
                                                xtitconnectd
benchd          0x01020306/tpid_osi_clts        BENCHD xtibenchd
```

Implementation Specifics

This file is used by XTI Name Server, part of **xti_api** software.

Files

/etc/xtiservices

Suggested Reading

Related Information

xtiprotocols File.

How to Manage XTI OSI Services, on page 3-14.

xtiserv Command.

xtiopts File

Purpose

XTI OSI Option Profiles Data Base.

Description

The **xtiopts** file contains sets of XTI options, with the following format:

```
option <set_option_name> [
 <{option_level, option_name, <NULL|option_value>*} [,]>*
]
```

where:

- `<set_option_name>` is a string of char (40 max).
- `<option_level>`, `<option_name>` and `<option_value>` are strings of char (40 max) or decimal values.

A set of options or **Option Profile** is identified by a `<set_option_name>` and each option within the profile is defined by:

`option_level`
: Level accessed by the option within the transport stack:
XTI_GENERIC, ISO_TP, INET_UDP, INET_TCP, INET_IP or X25_NP.

`option_name` the option name within the level.
Examples of option name are TCO_EXPD within the level ISO_TP or TCP_NODELAY within the level INET_TCP.

`option_value`
: Optional and represents the value for the option.

The options implemented in Bull-enhanced XTI are listed in Bull-enhanced XTI Option Profiles in Appendix C.

Examples

```
#-------------------------------------------------------
#
#       example_ltpdu:  Used by the example programs
#
#-------------------------------------------------------

option example_ltpdu [
        {ISO_TP,        TCO_LTPDU,      T_UNSPEC}
]
```

```
#-------------------------------------------------------------
#
#       connect_opt:
#       An example with option from different providers mixed.
#       This profile can be used with either
#               a TCP,UDP or OSI transport
#               at connection time (t_connect()).
#
#-------------------------------------------------------------

option connect_opt [
        {XTI_GENERIC,   XTI_DEBUG,      1,31},
        {XTI_GENERIC,   XTI_LINGER,     T_YES,  32},
        {ISO_TP,        TCO_EXPD,       T_YES},
        {ISO_TP,        TCO_PREFCLASS,  T_CLASS4},
        {ISO_TP,        TCO_ALTCLASS1,  T_CLASS2},
        {XTI_GENERIC,   XTI_SNDBUF,     3003},
        {XTI_GENERIC,   XTI_RCVBUF,     3004},
        {INET_TCP,      TCP_KEEPALIVE,  T_YES, 1},
        {XTI_GENERIC,   XTI_SNDLOWAT,   35},
        {XTI_GENERIC,   XTI_RCVLOWAT,   36},
        {INET_UDP,      UDP_CHECKSUM,   T_YES},
        {INET_IP,       IP_TTL,         30}
]


#-------------------------------------------------------------
#    Option of same level XTI_GENERIC to be used with t_optmgmt()
#-------------------------------------------------------------

option xti_level [
        {XTI_GENERIC,   XTI_DEBUG,      NULL},
        {XTI_GENERIC,   XTI_LINGER,     T_YES,  12},
        {XTI_GENERIC,   XTI_SNDBUF,     12},
        {XTI_GENERIC,   XTI_SNDBUF,     12}
]

option T_ALLOPT_alone [
        {XTI_GENERIC,   T_ALLOPT,       NULL}
]


#-------------------------------------------------------------
#       Numerical value can be used as well
#-------------------------------------------------------------

option xti_level_num [
        {XTI_GENERIC,   0x0001,         NULL},
        {0xffff,        XTI_LINGER,     T_YES},
        {0xffff,        0x1001,         12},
        {XTI_GENERIC,   XTI_SNDBUF,     12}
]


#-------------------------------------------------------------
#       Example for XX25 options
#-------------------------------------------------------------

option ex_xx25_opt [
        {X25_NP,        T_X25_CONN_DBIT,        T_YES},
        {X25_NP,        T_X25_PKTSIZE,  128,    128}
]
```

Implementation Specifics

This file is used by XTI Name Server, part of **xti_api** software.

Files

/etc/xtiopts

Suggested Reading

Related Information

XTI Option Profiles Configurator, on page 3-25,

xtiopt Command, on page 6-10,

t_optmgmt() Subroutine, on page 4-39,

Bull-enhanced XTI Options in Appendix C.

xtitrace and **xticnxtrace Files**

Purpose

XTI Trace Levels Data Base.

Default Trace Levels defined by the administrator and concerning:

- XTI library trace levels in **xtitrace** file.
- XTI library and kernel trace levels in **xticnxtrace** file.

Description

In **xtitrace** and **xticnxtrace** files, the trace levels value is made of 4 bytes set with the OR-combination of the defined values for each XTI trace level (XTI_LEVELxx), that is:

```
0xXXXXXXXX
```

where X is an hexadecimal digit.

It enables generation of trace in the XTI library for the **xtitrace** file and in both XTI library and kernel (**XTI4MOD** module) for the **xticnxtrace** file, if the user has not set other trace levels defining other trace files defined by the **XTI_TRACE_LEVEL** and **XTI_TRACE_LEVELCNX**

Example

To trace DATA TRANSFER functionnalities (**XTI_LEVEL30**) with Description of Entry and Return of external XTI Library Functions (**XTI_LEVEL11**) and of I/O parameters values (**XTI_LEVEL24**) the value set in **xtitrace** or **xticnxtrace** must be:

```
20800400
```

Implementation Specifics

This file is used by XTI Name Server, part of **xti_api** software.

Files

/etc/xtitrace
/etc/xticnxtrace

Suggested Reading

Related Information

How to Set XTI Administrative Trace Levels, on page 3-29.

How to Set XTI User Trace Levels, on page 3-33.

xtitracelevel Command, on page 6-8.

Appendix C. Options

List of Bull-enhanced XTI Options

XTI_GENERIC-level Options: Options for any Transport Provider

Option Name	Option Type	Legal Option Value	Meaning
XTI_DEBUG	struct t_deblevel	see Setting Trace Levels on page 4-45	set XTI Trace Levels
XTI_LINGER	struct linger	see on page 4-43	linger on close if data is present
XTI_RCVBUF	unsigned long	size in octets	receive buffer size
XTI_RCVLOWAT	unsigned long	size in octets	receive low-water mark
XTI_SNDBUF	unsigned long	size in octets	send buffer size
XTI_SNDLOWAT	unsigned long	size in octets	send low-water mark

The XTI_GENERIC-level options are fully described in **t_optmgmt ( )** subroutine, on page 4-39.

ISO_TP-level Options: Options for OSI COTS and NetShare (RFC 1006)

Supported Options for OSI				
Option Name	**Default Value**	**Allowed Values**	**Option Type**	**Option Meaning**
TCO_LTPDU	2048 if preferred class is 0, else 8192	128 256 512 1024 2048 4096 or 8192	unsigned long	Maximum length of TPDU
TCO_PREFCLASS	Class 4 by default, Class 2 for TNULLCLNP network service on WAN	T_CLASS0 T_CLASS2 T_CLASS3 T_CLASS4	unsigned long	Preferred class
TCO_ALTCLASS1	T_UNSPEC	T_CLASS0 T_CLASS2 T_CLASS3 T_CLASS4	unsigned long	First alternate class
TCO_EXTFORM	T_NO	T_YES T_NO	unsigned long	Extended format
TCO_FLOWCTRL	T_NO is preferred class 0, else T_YES	T_YES T_NO	unsigned long	Flow control
TCO_CHECKSUM	T_NO	T_YES T_NO	unsigned long	Checksum
TCO_PRIORITY	T_PRIDFLT	T_PRIDFLT T_PRILOW T_PRIMID T_PRIHIGH T_PRITOP	unsigned long	Priority
TCO_EXPD	T_NO if preferred class 0, else T_YES	T_YES T_NO	unsigned long	Expedited data
TCO_RCV_CRDT	15	1 to 15	unsigned long	Maximum receive credit
TCO_NETSRV	TFULLCLNP	TNULLCLNP TFULLCLNP TSPEE	unsigned long	Network service used
TCO_DEBUG	No level set	ERRTRACE CONTRACE XFERTRACE FCONTRACE FXFERTRACE ICBTRACE	unsigned long	Debug level
TCO_X25FAC	None	refer to ISO8208	unsigned char[]	X25 facilities
TCO_X25CALLDT	None	refer to ISO8208	unsigned char[]	X25 call data

The ISO_TP-level options are fully described in **Appendix A. ISO Transport Protocol Information** in *X/Open Transport Interface XPG4 CAE Specification Version 2*, except for the options TCO_RCV_CRDT, TCO_NETSRV, TCO_DEBUG, TCO_X25FAC and TCO_X25CALLDT which are part of Bull implementation specifics.

Supported Options for NetShare (RFC 1006)				
Option Name	**Default Value**	**Allowed Values**	**Option Type**	**Option Meaning**
TCO_LTPDU	64 K – 4		unsigned long	Maximum length of TPDU
TCO_EXPD	T_YES	T_YES T_NO	unsigned long	Expedited data

INET_TCP, INET_UDP and INET_IP-level: Options for TCP/IP and UDP

TCP/IP Communication Stack Supported Options					
Option Level	**Option Name**	**Default Value**	**Allowed Values**	**Option Type**	**Option Meaning**
INET_TCP	TCP_KEEPALIVE	(1) tcp_keepidle	(3)	struct t_kpalive	Checks if connections are alive
INET_TCP	TCP_MAXSEG	(1) tcp_mssdflt	(3)	unsigned long	Get TCP maximum segment size
INET_TCP	TCP_NODELAY	T_NO	(3)	unsigned long	Don't delay send to coalesce packets
INET_UDP	UDP_CHECKSUM	(1) udpchsum	(3)	unsigned long	Checksum computation.
INET_IP	IP_BROADCAST	T_NO (2)	(3)	unsigned int	Permit sending of broadcast messages
INET_IP	IP_DONTROUTE	(1) ipforwarding	(3)	unsigned int	Bypass the standard routing information
INET_IP	IP_OPTIONS	No option set	(3)	array of unsigned char	IP per-packet options
INET_IP	IP_REUSEADDR	T_NO	(3)	unsigned int	Allow local address reuse
INET_IP	IP_TOS	No service set	(3)	unsigned char	IP per-packet type of service
INET_IP	IP_TTL	(1) maxttl	(3)	unsigned char	IP per-packet time to live

Notes:

1. To obtain the default value, run the command **no –a** and read the corresponding variable, for instance **tcp_keepidle.**
2. For further information run the command **ifconfig xxx**, where **xxx** is the used interface.
3. The allowed values are listed in *AIX Performance Tuning Guide*.

The INET-level options are fully described in **Appendix B. Internet Protocol-specific Information** in *X/Open Transport Interface XPG4 CAE Specification Version 2*.

X25_NP-level Options: Options for XX25

Supported Options for XX25				
Option Name	**Default Value**	**Allowed Values**	**Option Type**	**Option Meaning**
T_X25_USER_DACK	T_NO	T_YES/T_NO	unsigned long	Explicit Acknowledgement of data with delivery bit
T_X25_USER_EACK	T_NO	T_YES/T_NO	unsigned long	Explicit Acknowledgement of expedited data
T_X25_RST_OPT	T_NO	T_YES/T_NO	unsigned long	Support of resets by user
T_X25_VERSION	(implementation defined)	T_X25_yyyy with ”yyyy” representing the year of the X.25 recommendation Read–only option	unsigned long	Version of ITU–T Recommendation X.25 or ISO/IEC 8208 (X.25)
T_X25_DISCON_REASON	0xf1	Reason specified in ISO/IEC 8208 (X.25)	unsigned long	Reason (cause and diagnostic) of a Connection release
T_X25_DISCON_ ADD		Read–only option	struct x25facaddr	Address of the user that released the connection
T_X25_D_OPT	(implementation defined)	T_YES/T_NO Read–only option	unsigned long	Support of the D bit
T_X25_CONN_DBIT	T_NO	T_YES/T_NO	unsigned long	Setting of the D–bit during the connection phase in order to negotiate the support of the D_bit during data transfer
T_X25_PKTSIZE	128	size in bytes from 16 to 4096, T_UNSPEC	struct x25facval	Packet Size
T_X25_WINDOWSIZE	2	size from 1 to 7 or from 1 to 127 (in extended format), T_UNSPEC	struct x25facval	Window Size
T_X25_TCN	9600	Throughput in bits/s from 75 to 192000, T_UNSPEC	struct x25facval	Througput Class Negotiation
T_X25_CUG	0	index from 0 to 9999, T_UNSPEC	unsigned long	CUG (Closed User Group)
T_X25_CUGOUT	0	index from 0 to 9999, T_UNSPEC	unsigned long	CUG with Outgoing Access
T_X25_BCUG	0	index from 0 to 9999, T_UNSPEC	unsigned long	Bilateral CUG
T_X25_FASTSELECT	T_NO	T_X25_FASTSEL_NOREST/ T_X25_FASTSEL_REST/ T_NO	unsigned long	Fast Select
T_X25_REVCHG	T_NO	T_YES/T_NO	unsigned long	Reverse Charging
T_X25_NUI		format determined by the network administration	string	NUI – Network User Identification
T_X25_CHGINFO_REQ	T_NO	T_YES/T_NO	unsigned long	Charging Information Service Request

Supported Options for XX25				
Option Name	**Default Value**	**Allowed Values**	**Option Type**	**Option Meaning**
T_X25_CHGINFO_MU		Read–Only Option	string	Charging Information: Monetary Unit
T_X25_CHGINFO_SC		Read–Only Option. One structure per tariff period	struct x25facval	Charging Information: Segment Count
T_X25_CHGINFO_CD		Read–Only Option. One structure per tariff period	struct x25facin-focd	Charging Information: Call Duration
T_X25_RPOA		Index of each RPOA from 0 to 9999	Arrray of unsigned longs	RPOA–Recognised Private Operating Agency
T_X25_CALLDEF		value of the reason code ('code' field): –T_X25_CLDEF1 –T_X25_CLDEF2 –T_X25_CLDEF3 –T_X25_CLDEF4	struct x25facaddr	Call Deflection Selection
T_X25_CALLRED		Read–Only Option value of the reason code ('code' field): –T_X25_CLDEF1 –T_X25_CLDEF2 –T_X25_CLDEF3 –T_X25_CLDEF4 –T_X25_CLRED1 –T_X25_CLRED2 –T_X25_CLRED3 –T_X25_CLRED4	struct x25facaddr	Call Redirection or Deflection Notification
T_X25_CALLADDMOD		value of the reason code –T_X25_CLDEF1 –T_X25_CLDEF2 –T_X25_CLDEF3 –T_X25_CLDEF4 –T_X25_CLRED1 –T_X25_CLRED2 –T_X25_CLRED3 –T_X25_CLRED4	unsigned long	Called Line Address Modified Notification
T_X25_TDSAI		Transit delay in milliseconds from 0 to 65534, T_UNSPEC	unsigned long	Transit Delay Selection and Indication
T_X25_CALLING_ADDEXT		Value for the address type ('addr_type' field): –T_X25_NSAPADDR –T_X25_OTHERADDR	struct x25addext	Calling Address Extension
T_X25_CALLED_ADDEXT		Value for the address type ('addr_type' field): –T_X25_NSAPADDR –T_X25_OTHERADDR	struct x25addext	Called Address Extension
T_X25_MTCN	9600	Minimum Throughput in bits/s from 75 to 192000, T_UNSPEC	struct x25facval	Minimum Throughput Class Negotiation
T_X25_EETDN		Transit Delay in milliseconds from 0 to 65534, T_UNSPEC	struct x25eetdn	End–to–End Transit Delay Negotiation

Supported Options for XX25				
Option Name	**Default Value**	**Allowed Values**	**Option Type**	**Option Meaning**
T_X25_PRIORITY		Value of the priority type ('typeval' field): –T_X25_PRIDATA –T_X25_PRIGAIN –T_X25_PRIKEEP Values of priority : ('targetval' and 'lowval' fields): –T_PRITOP –T_PRIHIGH –T_PRIMID –T_PRILOW –T_PRIDFLT –T_UNSPEC	struct x25facpr	Priority
T_X25_PROTECTION		Value of the protection type ('typeval' field): –T_X25_SRCPROTECT –T_X25_DESTPROTECT –T_X25_GLBPROTECT Values of protection ('targetval' and 'lowval' fields): –T_NOPROTECT –T_PASSIVEPROTECT –T_ACTIVEPROTECT	struct x25facpr	Protection
T_X25_EDN	T_NO	T_YES/T_NO/T_UNSPEC	unsigned long	Expedited Data Negotiation
T_X25_LOC_NONX25		buffer in raw form as encoded in the local non–X25 facilities part of the facilities field of the X.25 packet	string	Non–X25 local facilities
T_X25_REM_NONX25		buffer in raw form as encoded in the remote non–X25 facilities part of the facilities field of the X.25 packet	string	Non–X25 remote facilities
T_X25_REM_EQUILISTEN	T_NO	T_YES/T_NO	unsigned long	Support of equal distribution of incoming calls between listeners with the same criteria on the same address

Note: The T_X25_EQUILISTEN option allows to select the service of an equal distribtuion on the same address.
To be taken into account, this option has to be selected before the t_bind() call done for a listener. By selecting this option, the t_bind() function never returns the T_ADDRBUSY error.
This option allows to distribute incoming calls in an equal way between listeners having the matching criteria.
The distribution depends on the number of connections that have been already attributed to each listener having the matching criteria.

Appendix D. OSI Addressing

Bull-enhanced XTI and OSI Addressing

Note: Full description of OSI addressing and profiles is available in *OSI Services Reference Manual*: **OSI Addressing** Chapter and **OSI Profiles** Appendix.

XTI Functions and OSI Addressing

Addresses have to be used as parameters of these XTI functions:

t_bind Bind an address to a transport endpoint.

t_connect Establish a connection with another transport user.

t_listen Listen for a connect indication.

t_rcvconnect Receive the confirmation from a connect request.

t_bind needs a local address only, whereas **t_connect**, **t_rcvconnect** and **t_listen** need a remote and eventually part of a local address.

Note: In order to conform to *X/Open Transport Interface XPG4 CAE Specification Version 2* an address may be defined as input parameter for the **t_accept** function (Accept a connect request), but is not significant in this implementation.

A complete address is made of up to five address components defined by these types:

TNETSRV Network Service for **t_connect**, **t_rcvconnect** and **t_listen.**

TLSAP Local LSAP for **t_bind.**
Remote LSAP for **t_connect**, **t_rcvconnect** and **t_listen.**

TTSEL Local Transport SELector for **t_bind**.
Remote Transport SELector for **t_connect**, **t_rcvconnect** and **t_listen.**

TNSAP Local NSAP for **t_bind**.
Remote NSAP or local and remote SNPA for **t_connect**, **t_rcvconnect** and **t_listen** depending on the network service.

TLNSAP Local NSAP for **t_connect**, **t_rcvconnect** and **t_listen** depending on the network service.

XTI functions	Network Service	Local LSAP	Local TSEL	Local NSAP	Remote LSAP	Remote TSEL	Remote NSAP[1]
t_bind()		TLSAP	TTSEL	TNSAP			
t_connect()	TNSERV			TLNSAP[2]	TLSAP	TTSEL	TNSAP
t_listen()	TNSERV			TLNSAP[2]	TLSAP	TTSEL	TNSAP
t_rcvconnect()	TNSERV			TLNSAP[2]	TLSAP	TTSEL	TNSAP

Figure 10. XTI Functions and OSI Addresses Components

Notes:

1. Or local and remote SNPA.
2. Optionally specified to overwrite the local NSAP given during the **t_bind** call.

Addresses Format

Up to five address components may be concatenated in the address buffer.

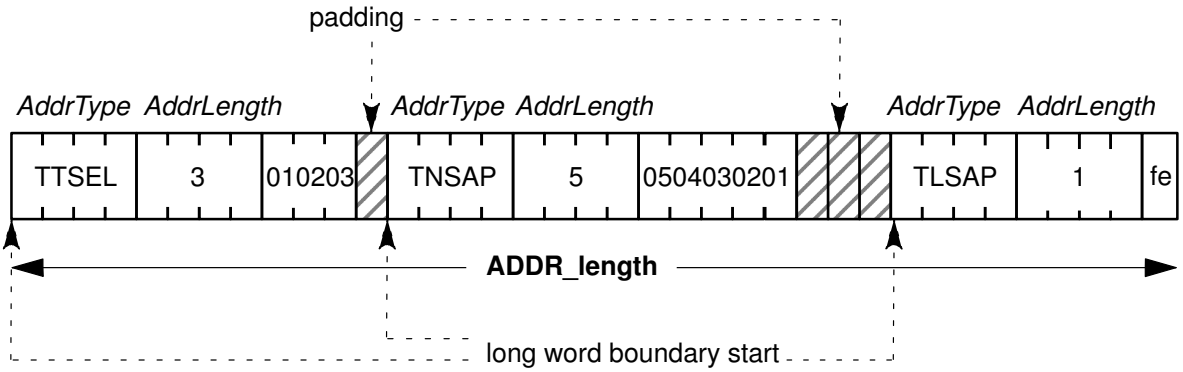


Figure 11. OSI Stack Address Structure Example

The addresses are specified in TLV format (Type–Length–Value).

Each component is specified by a header (type and length) possibly followed by a value.

```
typedef struct t_tladdr{
        unsigned long AddrType;
                /*type of address component */
        unsigned long AddrLength;
                /*length of address component in bytes */
}TLAddr_t;
```

Each address component must start at a long-word boundary.
The macro T_ALIGN defined in the include file **xti_api/xti.h** may be used to align TLV items within the address buffer.

The structure type *t_tladdr* and the other definitions necessary for definition of an OSI address are provided in the include file **sys/osi/osi1to4.h**.

OSI Addresses Components

NETSRV: Network Service

- Network Service for **t_connect**, **t_rcvconnect** and **t_listen**

  NETSERV is optional in a **t_connect** address (if not specified, **TFULLCLNP** is the default value)

- Length: 4 bytes
- Values:
  - **TFULLCLNP,** ISO Internet
  - **TNULLCLNP,** Null ISO Internet on LAN or CONS on WAN
  - **TSPEE,** ISO Internet on LAN, CONS on WAN

TLSAP: Link Service Access Point

- Local LSAP for **t_bind**

  Remote LSAP for **t_connect**, **t_rcvconnect** and **t_listen.**

- On incoming connection, TLSAP is not significant (no check is done by the OSI transport between the bounded LSAP and the LSAP on which the incoming connection arrives).
- On an outgoing connection request **t_connect,** the LSAP is meaningful only when **TNETSERV** equals to **TNULLCLNP**.

TLSAP	t_bind	t_connect, t_listen, t_rcvconnect
OSI_LSAP_DSA	DSA : Non–full OSI conformance	DSA : Non–full OSI conformance
OSI_LSAP_OSI	OSI : Full OSI conformance	OSI : Full OSI conformance
OSI_LSAP_ANY	On incoming connections: meaningless	**Not allowed**
	On outgoing connections, if Null Internet is selected on LAN, the remote LSAP will be used as local LSAP, else OSI_LSAP_OSI will be used	

Figure 12. TLSAP Meaning

TTSEL: Transport SELector

- Local Transport SELector for **t_bind**

  Remote Transport SELector for **t_connect**, **t_rcvconnect** and **t_listen**

- Length: 32 bytes max
- A wildcard value is defined: **OSI_TSEL_ANY**

TTSEL Wildcard Value	t_bind	t_connect, t_listen, t_rcvconnect
OSI_TSEL_ANY	On incoming connections: a listening endpoint bound with **OSI_TSEL_ANY** will receive all the connect indications for which no exact TTSEL match exists	Used as any other remote TTSEL No exact match occurs on **t_listen**
	On outgoing connections: used as any other calling TTSEL	

Figure 13. TTSEL Wildcard Value Meaning

TNSAP

- Local NSAP (Network Service Access Point) for **t_bind**

  Remote NSAP or local and remote SNPA for **t_connect**, **t_rcvconnect** and **t_listen** depending on the network service:

  – if **TFULLCLNP** or **TSPEE**, remote NSAP (Network Service Access Point)

  – if **TNULLCLNP**, local and remote SNPA (Sub-network Point of Attachment)

- Length: 20 bytes max

0xff	LAN_LLC1	Length in bytes	Local mac @	Length in bytes	Remote mac @
1 byte	1 byte	1 byte	6 bytes	1 byte	6 bytes

or:

0xff	WAN_SVC	Length in half bytes	Local x121 @	Length in half bytes	Remote x121 @
1 byte	1 byte	1 byte	8 bytes	1 byte	8 bytes

or:

0xff	WAN_PVC	Length in half bytes	Local PVC name
1 byte	1 byte	1 byte	8 bytes

Figure 14. TNSAP for t_connect, t_listen and t_rcvconnect on TNULLCLNP (Sub-network Point of Attachment)

- A wildcard value is defined: **OSI_NSAP_ANY**

TNSAP Wildcard Value	**t_bind**	**t_connect, t_listen, t_rcvconnect**
OSI_NSAP_ANY	On incoming connections: a listening endpoint bound with **OSI_NSAP_ANY** will receive all the connect indications for which no exact TNSAP match exists	**Not allowed**
	On outgoing connections: the default local NSAP will be used as calling NSAP	

Figure 15. TNSAP Wildcard Value Meaning

TLNSAP

- Local NSAP for **t_connect**, **t_rcvconnect** and **t_listen**, used only when **TNETSRV** equals to **TFULLCLNP**, in order to:
  - indicate the local address an incoming connection has been passed on (**t_listen**). This enables the transport user to know the real NSAP the connect indication comes on when a wildcard address has been used in **t_bind**,
  - indicate to overwrite the local address previously given in **t_bind** for outgoing connection request (**t_connect**). This is useful especially when **OSI_NSAP_ANY** has been used at t_bind() to overwrite the local NSAP automatically generated, which may be not correct (according to route configuration) to access the remote NSAP given in TNSAP.
- Length: 20 bytes max

Wildcarding

Wildcard value is defined for each address component. Its behaviour is described with the address component.

The following rules solve the competition cases occurring when several listening endpoints use wildcard addresses:

- first, equality has priority over wildcarding,
- secondly, TSEL equality has priority over wildcarding.

Which may be summarized on the following example:

If an incoming connection arrives, calling **TSEL** XX and **NSAP** YY, at most the following four listening endpoints, listed in descendant order, may be bound to addresses matching the called address:

```
listening endpoint 1: bounded address: tsel value = XX, nsap value = YY
listening endpoint 2: bounded address: tsel value = XX, nsap value = OSI_NSAP_ANY
listening endpoint 3: bounded address: tsel value = OSI_TSEL_ANY, nsap value = YY
listening endpoint 4: bounded address: tsel value = OSI_TSEL_ANY,
                                          nsap value = OSI_NSAP_ANY
```

Network Type and OSI Addressing

This table makes out the correspondence between:

- the Network Type, parameter of an OSI Host as defined by the XTI Name Server. (Refer to **xtihost** command, on page 6-2)
- the OSI address

Network Type	TNETSRV	TLSAP	TNSAP
CONS/**WAN**/**SVC**:COTS over CONS on S VC	TNULLCLNP		0xff, WAN_SVC, local and remote X121 addresses
CONS/**WAN**/**PVC**:COTS over CONS on PVC	TNULLCLNP		0xff, WAN_PVC, PVC name
I_CLNS/**LAN**:COTS over inactive CLNS (Non–full OSI conformance) with Data Link Service Access Point = **DSA** For example Ethernet, Token Ring, FDDI	TNULLCLNP	OSI_LSAP_DSA	0xff, LAN_LCC1, local and remote MAC addresses
I_CLNS/**LAN**:COTS over inactive CLNS (Full OSI conformance) with Data Link Service Access Point = **OSI**	TNULLCLNP	OSI_LSAP_OSI	0xff, LAN_LCC1, local and remote MAC addresses
CLNS:COTS and CLNS over LAN and WAN (Full OSI conformance)	TFULLCLNP	OSI_LSAP_OSI	Remote NSAP
SPEE:COTS over CLNS (on LAN) or COTS over CONS (on WAN)	TSPEE	OSI_LSAP_OSI	Remote NSAP

Appendix E. X.25 Addressing

XTI/XX25 Functions and X.25 Addressing

Addresses have to be used as parameters of these XTI/XX25 functions:

t_bind Bind an address to an X.25 endpoint.

t_connect Establish a connection with another X.25 user.

t_listen Listen for a connect indication.

t_rcvconnect Receive the confirmation from a connect request.

Note: In order to conform to *X/Open Transport Interface XPG4 CAE Specification Version 2* an address may be defined as input parameter for the **t_accept** function (Accept a connect request), but is not significant in this implementation.

Note: Wildcard mechanism is not supported in this implementation.

An XX25 address is made up of several address components defined by these types:

X25CLDADDR X.25 Called Address.
Local X.121 for **t_bind** and **t_listen**.
Remote X.121 for **t_connect**.

X25CLGADDR X.25 Calling Address.
Local X.121 for **t_connect**.
Remote X.121 for **t_bind** and **t_listen**.

X25RSPADDR X.25 Responding Address.
Remote X.121 for **t_rcvconnect**.

X25UDATA Subsequent Application Identifier (SAI) for **t_bind**, **t_listen** and **t_connect.**

X25PVC X.25 PVC number for **t_bind** and **t_connect.**

XTI/XX25 functions	Local X.121 @	Remote X.121 @	SAI	PVC Number
t_bind()	X25CLDADDR	X25CLGADDR	X25UDATA	X25PVC
t_connect() **t_listen()** **t_rcvconnect()**	X25CLGADDR[1] X25CLDADDR	X25CLDADDR X25CLGADDR X25RSPADDR	X25UDATA X25UDATA	X25PVC

Figure 16. XTI/XX25 Functions and XX25 Addresses Components

Notes:

1. Optionally specified to overwrite the local address given during the **t_bind** call.

Addresses Format

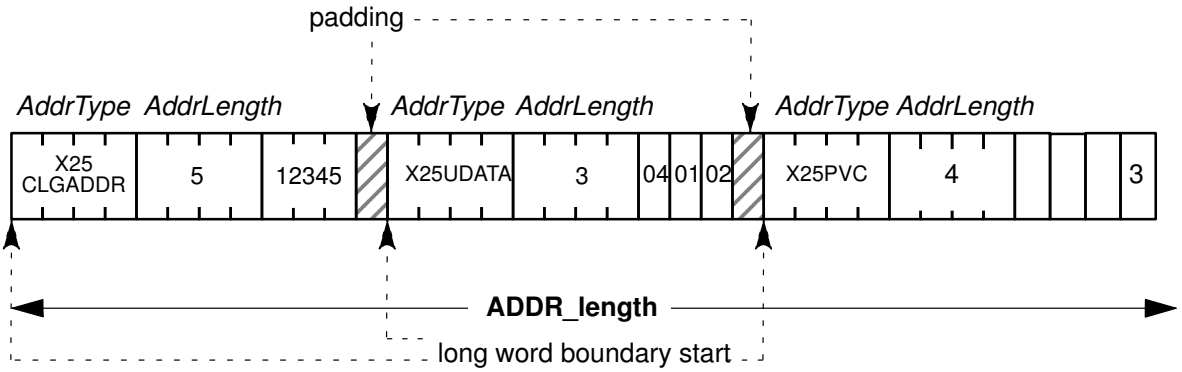


Figure 17. XX25 Address Structure Example

The addresses are specified in TLV format (Type–Length–Value).

Each component is specified by a header (type and length) followed by a value.

```
typedef struct X25Addr{
        unsigned long AddrType;
                /*type of XX25 address component */
        unsigned long AddrLength;
                /*length of XX25 address component */
}X25Addr_t;
```

Each address component must start at a long-word boundary.
The macro T_ALIGN defined in the include file **xti_api/xti.h** may be used to align TLV items within the address buffer.

The structure type *X25Addr* and the other definitions necessary for definition of an XX25 address are provided in the include file **xti_xx25/xx25addr.h**

XX25 Adresses Components

AddrType	AddrLength	Maximum Length	Type
X25CLDADDR	in half-bytes	X25MAXADDRLEN	X.121 address
X25CLGADDR	in half-bytes	X25MAXADDRLEN	X.121 address
X25RSPADDR	in half-bytes	X25MAXADDRLEN	X.121 address
X25UDATA	in bytes	X25MAXUDATALEN	set of bytes
X25PVC	in bytes: size of (unsigned long)	size of (unsigned long)	unsigned long

Glossary

Definitions

The following terms apply to the X/Open Transport Interface:

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

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

Client
The transport user in connection–mode who initiates the establishment of a transport connection.

Connection establishment
The phase in connection–mode that enables two transport users to create a transport connection between each other.

Connection–mode
A circuit–oriented mode of transfer in which data are passed from one user to another through an established connection in a reliable, sequenced manner.

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

Connection release
The phase in connection–mode that terminates a previously established transport connection between two users.

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

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

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

Expedited transport service data unit
Amount of expedited user data which preserves the user's identity from one end of a transport connection to the other (that is, an expedited message).

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

MAC Address
Medium Access Control Address

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

Peer user
The user with whom a given user is communicating above the X/Open Transport Interface.

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

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

Service primitive
A unit of information passed through a service interface that contains either a service request or service indication.

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

Socket.
(1) A unique host identifier created by the concatenation of a port identifier with a TCP/IP address. (2) A port identifier. (3) A 16-bit port number. (4) In NCS, a port on a specific host; a communications end point that is accessible through a protocol family's addressing mechanism. A socket is identified by a socket address. See also *socket address*.

STREAMS.
A kernel mechanism that supports development of network services and data communication drivers. It defines interface standards for character input and 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.

Synchronous execution
The mode of execution in which XTI routines may block while waiting for specific asynchronous events to occur.

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

Transport connection
The communication circuit established between two transport users in connection–mode.

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

Transport Interface
The library routines and state transition rules supporting the services of a transport protocol.

Transport provider
The transport protocol that provides to XTI the services of the transport layer.

Transport service data unit
The amount of user data whose boundaries are preserved from one end of a transport connection to the other (that is, a message).

Transport user
The user–level application or protocol that accesses the services of XTI.

Virtual circuit
A transport connection established in connection–mode.

X.25
A recommendation of the CCITT which defines the interface between a Data Terminal Equipment (DTE) and a Data Circuit terminating Equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuits (ISO 8208).

Acronyms

The following acronyms are used throughout this guide:

API Application Program Interface

BSD Berkeley Software Distribution

CLNS Connection-less Network Service

CLTS Connection-less Transport Service

CONS Connection-oriented Network Service

COTP Connection-oriented Transport Protocol

COTS Connection-oriented Transport Service

DSA Distributed System Architecture

ETSDU Expedited Transport Service Data Unit

LSAP Link Service Access Point

MAC Medium Access Control

NSAP Network Service Access Point

PVC Permanent Virtual Circuit

SAI Subsequent Application Identifier

SMIT System Management Interface Tool.

SNPA Sub-Network Point of Attachment

SVC Switched Virtual Circuit

TCP Transmission Control Protocol

TSAP Transport Service Access Point

TSEL Transport SELector

TSDU Transport Service Data Unit

UDP User Data Protocol

XTI X/OPEN Transport Interface

XX25 X.25 Programming Interface using XTI

Index

U

W

X

Vos remarques sur ce document / Technical publication remark form

Titre / Title : Bull DPX/20 XTI/XX25 Administrator & User Guide

Nº Reférence / Reference Nº : 86 A2 04AP 02

Daté / Dated : June 1996

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : _____ Date : _____

SOCIETE / COMPANY : _____

ADRESSE / ADDRESS : _____

_____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

Bull Electronics Angers S.A.
CEDOC
Atelier de Reprographie
331 Avenue Patton
49004 ANGERS CEDEX 01
FRANCE

PLACE BAR CODE IN LOWER LEFT CORNER

Bull Electronics Angers S.A.
CEDOC
Atelier de Reprographie
331 Avenue Patton
49004 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 04AP 02

Bull

Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

DPX/20
AIX
XTI/XX25
Administrator &
User Guide

86 A2 04AP 02

DPX/20
AIX
XTI/XX25
Administrator &
User Guide

86 A2 04AP 02

DPX/20
AIX
XTI/XX25
Administrator &
User Guide

86 A2 04AP 02