6

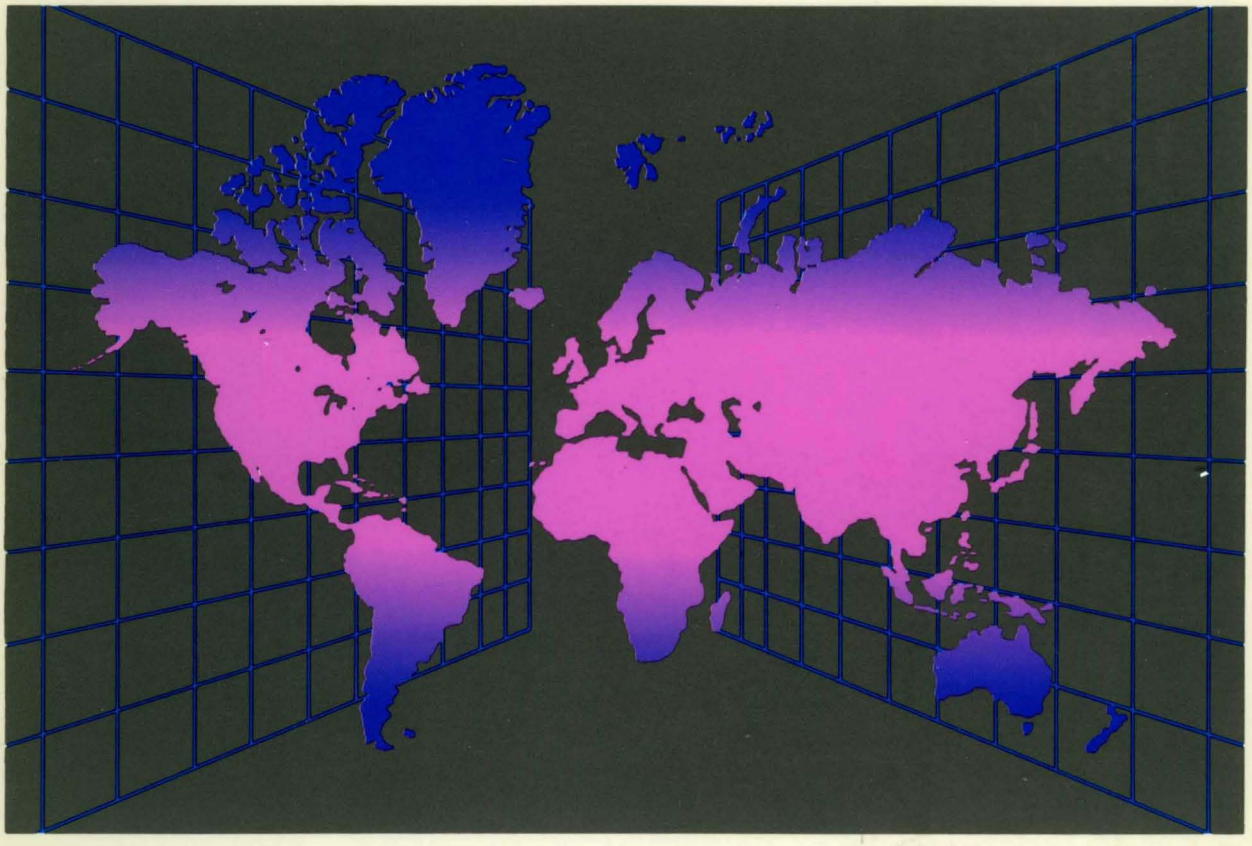
CTOS/Open Application Programming Interface Specification

Networking Services

CTOS open

Application Programming Interface Specification

Networking Services



PRENTICE HALL

CTOS/Open Application Programming Interface Specification

Networking Services

Prentice Hall
Englewood Cliffs, New Jersey 07632

Author: *Bert Miller*
Cover design: *April Bishop*
Editors: *Carol Collins, Gregory Viau, Tamara Westen*
Page design: *Milena Martin-Arana*
Production editing: *Phyllis McCrea*
Editorial/production supervision: *Harriet Tellem*
Prepress buyer: *Mary McCartney*
Manufacturing buyer: *Susan Brunke*
Acquisitions editor: *Karen Gettman*
Editorial Assistant: *Connie Uccelli*

Published by Prentice-Hall, Inc.
A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

The publisher offers discounts on this book when ordered in bulk quantities.
For more information, write: Special Sales/Professional Marketing, Prentice-Hall, Inc.,
Professional & Technical Reference Division, Englewood Cliffs, New Jersey 07632.

Draft 1.0 (June 1991). This document was produced using the Document Designer Series.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-194655-2

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

About This Manual ix

1 Overview
What Is CTOS/Open? 1-1
The OSI Reference Model 1-2
CTOS/Open Networking Services 1-5
Status of This Specification 1-6

2 Protocal Manager
Protocol Manager Overview 2-1
Parameter Definition File (PDF) 2-1
Protocol Manager Services 2-2
RegisterServiceProvider 2-3
DeregisterServiceProvider 2-6
RegisterServiceClient 2-8
DeregisterServiceClient 2-11
RequestServiceProvider 2-13
QueryProtocolManager 2-16
UpdateProtocolManager 2-19
Protocol Manager Status Data Structures 2-21
Service Provider Descriptor Block (SPDB) 2-21
Service Client Descriptor Block (SCDB) 2-22

3 Link Layer

Link Layer Overview 3-1
Parameter Definition File for Link Layers 3-3
Link Layer Name 3-4
Link Layer Services 3-4
OpenStationLL 3-5
OpenStationSL 3-8
CloseStation 3-11
ReadDLFrame 3-13
WriteDLFrame 3-17
DirectStation 3-21
DirectLink 3-24
Link Layer Status Data Structures 3-27
Generic Statistical Header Block (GHB) 3-27
DirectStation Commands 3-29
Link Layer Independent Commands 3-29
Link Layer Dependent Commands 3-33
DirectLink Commands 3-35
Link Layer Independent Commands 3-35
Link Layer Dependent Commands 3-36

4 Transport Layer

Transport Layer Overview 4-1
References 4-5
Explanatory Notes 4-6
Transport Endpoints 4-6
Transport Providers 4-6
Association of a Process to an Endpoint 4-7
Use of Same Protocol Address 4-8
Modes of Service 4-8
Error Handling 4-9
Synchronous and Asynchronous Execution Modes 4-9
Events and t_look() 4-12
Event Management 4-12

XTI Overview 4-14
Overview of Connection-Oriented Mode 4-14
Initialization/Deinitialization Phase 4-15
Overview of Connection Establishment 4-16
Overview of Data Transfer 4-19
Overview of Connection Release 4-22
Overview of Connectionless Mode 4-24
Initialization/Deinitialization Phase 4-24
Overview of Data Transfer 4-25
Mandatory XTI Features 4-27
Optional XTI Features 4-28
XTI Functions Versus Protocols 4-29
States and Events in XTI 4-31
Transport Interface States 4-31
Outgoing Events 4-33
Incoming Events 4-35
Transport User Actions 4-37
State Tables 4-37
Events and TLOOK Error Indication 4-40
Transport Protocol-Specific Options 4-41
Connection Mode Options 4-42
Connectionless Mode Options 4-42
CTOS Support of XTI procedures 4-43
Support of Optional Procedures 4-43
Support of Multiple Procedures by a Single Request 4-44
Use of Multiple Requests by a Single Procedure 4-45
Parameter Definition File for the Transport Layer 4-45
XTI Library Functions and Parameters 4-46
t_accept 4-47
t_bind 4-53
t_close 4-60

t_connect 4-63
t_getinfo 4-70
t_getstate 4-76
t_listen 4-80
t_look 4-85
t_open 4-89
t_optmgmt 4-97
t_rcv 4-102
t_rcvconnect 4-106
t_rcvdis 4-111
t_rcvrel 4-116
t_rcvudata 4-119
t_rcvuderr 4-125
t_snd 4-128
t_snddis 4-134
t_sndrel 4-138
t_sndudata 4-141
t_sync 4-147
t_unbind 4-152
Using the Transport Layer Interface 4-155
Transport Layer Interface Sequence of Functions 4-155
Example in Connection-Oriented Mode 4-156
Example in Connectionless Mode 4-158
Writing Protocol Independent Software 4-159

Appendixes
A Error Codes A-1
B Link API Event Codes B-1
C Transport API Definitions C-1
D ISO Transport Protocol Information D-1
E Internet Transport Specific Information E-1

Index I-1

List of Figures

1-1. OSI Reference Model 1-4

3-1. Link Layer 3-2

4-1. Transport Layer 4-3

4-2. Transport Interface States 4-32

4-3. Transport Interface Outgoing Events 4-34

4-4. Transport Interface Incoming Events 4-36

4-5. Transport Interface User Actions 4-37

4-6. Initialization/De-initialization State Table 4-38

4-7. Data-Transfer State Table for Connectionless-mode Service 4-38

4-8. Connection/Release/Data-Transfer State Table for Connection-mode Service 4-39

List of Tables

ATM-1 CTOS Variable-Naming Convention xi

Purpose of This Manual

This manual describes the procedural interface for each of the procedures in the CTOS/Open Standard for Networking Services. This manual is a companion volume to the other books in the CTOS/Open documentation set. This CTOS/Open Standard defines a common subset of application interfaces that is consistent across all versions of CTOS. Application software that conforms to the CTOS/Open Standard will run on CTOS platforms from multiple vendors that support the standard.

Note that the CTOS/Open Standards are *specifications* that are implemented by several CTOS-based operating systems. The Standard alone is not an operating system.

Audience

This specification is directed to software developers and Independent Software Vendors (ISVs) who will write networking applications that conform to the CTOS/Open Standard. It assumes that the reader has experience writing such applications under CTOS or under another operating system.

Related Documentation

In addition to this manual, the following books are included in the documentation set for CTOS/Open:

CTOS/Open Application Programming Interface Specification. This document describes the procedural interface definitions for each of the procedures supported by the CTOS/Open standard. It defines a common set of application interfaces that is consistent across all CTOS/Open-compliant versions of CTOS. The specification is intended for use by software developers and independent software vendors (ISVs) who want to write applications that conform to the CTOS/Open standard. It assumes that the reader has experience writing applications under CTOS or under another operating system.

CTOS/Open Application Programming Interface Specification: Printing Services. This document describes how to write applications that use the Generic Print System (GPS) or the Generic Print Access Method (GPAM).

CTOS/Open Application Programming Interface Specification: Computer Graphics Interface (CGI). This document describes each CGI operation and explains how to write CGI programs.

CTOS/Open Application Programming Interface Specification: Graphical User Interface (GUI). This document introduces XVT (Extensible Virtual Toolkit) and describes its relationship to other windowing interfaces. It also reviews the current XVT operations set, which includes functions, macros, constants, and types. This specification is intended for programmers who want to write applications that run in different window environments and work across all CTOS-based platforms. This document is a preliminary working draft. The final draft will be available some time after the release of XVT on CTOS.

CTOS/Open Programming Practices and Standards: Application Design. This document describes practices and standards programmers should follow to ensure that their applications are portable between implementations of CTOS. Following the recommendations in the standard can greatly simplify porting an application from one CTOS-based operating system to another. This manual can also serve as a "programmer's primer" for those new to CTOS/Open. It provides many programming examples.

CTOS/Open Programming Practices and Standards: User Interface Design. This document provides guidelines for designing graphical user interfaces for CTOS-based applications. It contains information about the user-interface standard called Common User Access (CUA), which has become part of the public domain. This document is a working draft.

Exploring CTOS. This book, available from Prentice Hall, provides an excellent overview of the CTOS operating system and the fundamentals of message based operating systems, illustrating how they make a highly efficient platform for distributed applications.

Organization

This manual is organized as follows:

- Chapter 1 introduces you to the purpose and definition of the CTOS/Open Standard for Networking Services.
- Chapter 2 presents the procedures for the CTOS/Open Protocol Manager.
- Chapter 3 presents the procedures for the CTOS/Open Link Layer Interface.
- Chapter 4 presents the procedures for the CTOS/Open Transport Layer Interface.
- Appendix A describes the error codes for the networking services.
- Appendix B describes the event codes for the networking services Link Layer Interface.
- Appendix C describes the event codes and other information for the networking services Transport Layer Interface.

The CTOS Naming Convention

The examples in this book follow a specific naming convention, which is designed to improve the readability of source code. This naming convention incorporates explanatory prefixes and suffixes on all variables and procedure names. The variable names should also be explanatory. These conventions are particularly important when programming in a language like C, which tends toward cryptic syntax.

Each variable takes the form <Prefix><Root>Name<Suffix>. Prefix, root, and suffix do not all have to be present. In fact, most variables do not use the suffix. Table ATM-1 describes the CTOS naming convention.

For example, the following variables define a data buffer:

pBuffer. A pointer to the start of the buffer.

sBufferMax. The maximum size of the buffer.

sBufferDataRet. The size of the data actually written to the buffer, returned by the procedure that writes to the buffer.

psBufferDataRet. Address of sBufferDataRet. Passed to the procedure that writes to the buffer, telling that procedure where to return the value of sBufferDataRet.

Table ATM-1. CTOS Variable-Naming Convention

(Page 1 of 2)

Token	Meaning
PREFIXES:	
b	Byte. A character or unsigned 8-bit integer.
c	Count. A two-byte unsigned integer.
f	Flag. A one-byte flag. True = 0FFh, False = 0.
i	Index. A two-byte unsigned integer.
l	Literal. A constant.
n	Number. A two-byte unsigned integer. Same as Count.
o	Offset. A two-byte offset from a segment base address.
p	Pointer. A logical memory address. Consists of a two-byte segment identifier and a two-byte offset.
q	Quad. A four-byte unsigned integer.
rg	Array. Usually used with another prefix. For example, the prefix "rgb" identifies an array of bytes.
s	Size. A two-byte unsigned integer.
sb	String. An array of bytes where the first byte is the size of the string.
w	Word. A two-byte integer.
cb	Count of bytes.
pb	Pointer to a string of bytes.

continued

Table ATM-1. CTOS Variable-Naming Convention

(Page 2 of 2)

Token	Meaning
ROOTS:	
erc	Two-byte status code.
exch	Two-byte exchange number.
fh	Two-byte file handle.
lfa	Four-byte logical file address.
ra	Two-byte relative address. Synonymous with offset.
rq	Request block. Size varies.
sa	Two-byte segment identifier.
sn	Selector. Segment identifier for a protected-mode memory address.
sr	Paragraph number. Segment identifier for a real-mode memory address.
userNum	Two-byte user number.
SUFFIXES:	
Last	Largest allowable index in an array.
Max	Maximum size of an array or buffer (Max = Last + 1).
Ret	Indicates a variable whose value is set by a called procedure, and returned to the current one.

1
Overview

What Is CTOS/Open?

The CTOS/Open Advisory Council (CTOS/Open for short) was formed as a joint effort among manufacturers, resellers, distributors, software developers, hardware developers, and users to establish and promote the CTOS-based architecture as a standard for distributed network computing.

The aim of CTOS/Open is to increase the number of CTOS-based applications and to maximize the return on investment in software development for independent vendors and users. CTOS/Open sets portability standards for the CTOS operating system and its variants, and integrates evolving standards into a common, beneficial, and continuing strategy.

The CTOS/Open Application Programming Interface (API) Specification defines a set of procedural interfaces for the CTOS operating system. This common CTOS interface definition ensures portability of application software across CTOS platforms from multiple vendors. In addition, it provides direction for future development by defining a consistent migration path for software. This approach enables new technological advances to coexist with earlier developments, and protects costly investments made in software development.

The OSI Reference Model

ISO defines an architectural model, called the Open Systems Interconnection Reference Model, that provides a common basis for the coordination of standards. These standards enable different types of computer equipment (for example, mainframes and workstations) from different manufacturers to interconnect and communicate. This model also provides a common reference for maintaining consistency for all standards, protocols, and services for computer communications as well as a functional framework for development of new standards. The CTOS/Open API for Networking Services is based on the lower layers of the OSI model.

The OSI Reference Model represents a seven-layer architecture as a "stack." Each layer uses the services of the layer below and provides services to the layer above. All seven layers have peer-to-peer protocols that allow services of each layer to communicate with services of the same layer on another system.

Grouping protocols by layer allows flexibility and modularity. Each layer of the OSI model groups functions logically:

- Layer One, the Physical Layer, defines physical transfer of information between nodes. It specifies terminal interfaces, electrical characteristics, and mechanical connections. It also specifies functions and procedures for using and maintaining a physical connection for bit transmission between two data-link entities.

- Layer Two, the Data Link Layer, controls the point-to-point transfer of information over the physical link between two network entities. It manages the link connection supervising data interchange, synchronizing and delimiting. It also manages frame sequencing, link flow control, error detection and recovery at the Physical Layer, and identification and parameter exchange.

- Layer Three, the Network Layer, switches and routes information from point-to-point through a connected group of systems. It controls routing, relaying, and network connections. It also controls logical channel control, segmenting and blocking, error detections and recovery, sequencing and flow control.

- Layer Four, the Transport Layer, specifies end-to-end data integrity and quality of service. It also provides Transport to Network address mapping, multiplexing, end-to-end error detection and recovery control, flow regulation, and end-to-end segmentation and sequencing of data units.

- Layer Five, the Session Layer, coordinates interaction and dialogue between Presentation entities. It provides administration services such as establishing and releasing session connections, and dialogue services such as data exchange, interaction management and synchronization, and recovery from Transport connection failure.

- Layer Six, the Presentation Layer, allows an application to properly represent and interpret the data being communicated. It specifies data interpretation, transformation, formatting, structuring, and syntax selection. This layer is responsible for conversion between standard representations of data and the format unique to a particular system.

- Layer Seven, the Application Layer, allows the application processes to gain access to OSI communication services, and to communicate with their application process partners by means of Application entities and protocols. The Application Layer provides services such as file transfer, security and access control, remote job entry, job initiation and termination, synchronizing cooperating applications, and message transfer system management function, as well as application management.

Each OSI layer described in Figure 1-1 provides services to its users in the layer above by using functions available within that layer and the services of the layers below. At each layer boundary, the lower layer is the provider of a service, or *service provider*. The upper layer is the user, or client, of the service, and is called a *service client*.

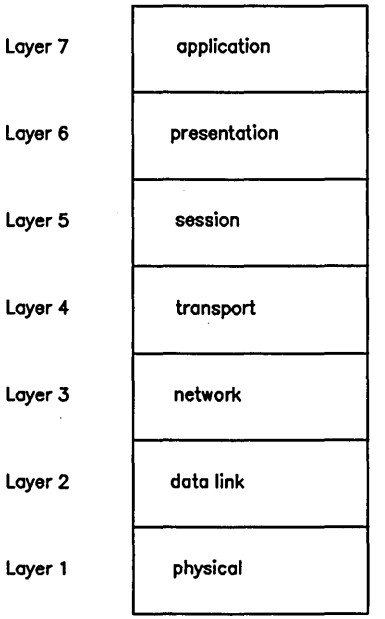


Figure 1-1. OSI Reference Model

CTOS/Open Networking Services

The CTOS/Open Advisory Council introduced the *CTOS/Open Application Programming Interface Specification: Networking Services* to increase the number of networking applications available for CTOS-based systems. These networking services are based on existing standard APIs (such as X/Open) where such standards exist. These networking services, although modeled after the layers of the OSI reference model, will be independent of any specific underlying protocol, and can be implemented by products which use non-OSI protocols.

Initially, interfaces are defined for the OSI Data Link Layer and the OSI Transport Layer. These are the layers where standardization on an open API will most benefit CTOS vendors and users today, by increasing application portability to and from non-CTOS platforms, by facilitating simultaneous application development for CTOS and non-CTOS platforms, and by permitting the development by independent vendors of applications which will operate with CTOS platforms from multiple vendors and with applications from other independent vendors.

Future versions of this specification will include additional interfaces.

This CTOS/Open specification is not itself an operating system. It is a definition of permissible procedural interfaces to a CTOS/Open compliant version of CTOS. Porting an application that conforms to the CTOS/Open Standard from one CTOS-based operating system to another should require, at most, relinking the program with the set of library procedures on the target system.

The APIs described in this document may be implemented in products sold separately from the CTOS-based operating system. Consult each product's documentation to determine whether or not the product implements this specification.

For brevity, the standard CTOS/Open API that is defined in this specification is referred to simply as *the Standard*, or *the CTOS/Open Standard*.

Status of This Specification

This revision of the *CTOS/Open Applications Programming Interface: Networking Services* is Draft 1.0. Draft 2.0 is not expected to include changes to the API for the Link Layer and the Protocol Manager functions related to the Link Layer. Some minor changes are expected in the future to the API for the Transport Layer and the Protocol Manager functions related to that layer.

2
Protocol Manager

Protocol Manager Overview

This document, in subsequent sections, will describe several CTOS request interfaces without specific request codes. The Protocol Manager is the means of dynamically informing Service Clients which request codes to use to access the services of their target Service Providers.

The Protocol Manager is a system service which allows a connection to be established between Service Providers and Service Clients for device independence. The role of the Protocol Manager includes four key areas:

- Providing directory services to Service Clients by returning address information for Service Providers (fully qualified device specifications)
- Giving out Service Provider request codes to Service Clients
- Tracking active Service Providers and Service Clients to provide current configuration information
- Facilitating parameter passing from a Service Client to a Service Provider

Parameter Definition File (PDF)

A properly written Service Client ought to be able to use the services of many different Service Providers. Similarly, a well-written Service Provider ought to be able to provide services to many different Service Clients. To accomplish these goals, each Service Client must be able to pass to the Service Provider specific parameters describing the service desired. These parameters often are specific to each Service

Provider/Service Client pair, so they ought not be hard-coded into the Service Client.

The Protocol Manager provides for all these needs by the use of Parameter Definition Files (PDF). Service dependencies are kept in a Parameter Definition File rather than in the Service Client or Service Provider themselves. The parameters in a PDF are dependent upon the type of the Service Provider. Every parameter in the file is terminated by a new line character (hexadecimal 0A). Comments delimited by a pair of colons are allowed in the string. Optionally, a string itself can be directly specified in the *OpenStationSL*, *OpenStationLL*, or *t_open* request; in this case, the same rules must be followed.

The only required information is the Service Provider Name (excluding the Device Specification which is supplied by the Protocol Manager), which is the first entry in the file or string. Any comments in the PDF must be deleted prior to passing the PDF string in the opening call (*OpenStationSL*, *OpenStationLL*, or *t_open*). This function will be done by the Protocol Manager if the PDF is a file.

See the Link Layer and Transport Layer sections for more details on the use of Parameter Definition Files for these layer boundaries.

Protocol Manager Services

The Protocol Manager supports the following requests:

RegisterServiceProvider
DeregisterServiceProvider
RegisterServiceClient
DeregisterServiceClient
RequestServiceProvider
QueryProtocolManager
UpdateProtocolManager

Description

A Service Provider makes the RegisterServiceProvider call to the Protocol Manager to declare its name and the set of request codes it serves. For instance, a Link Layer makes the RegisterServiceProvider call to enable Link Clients to locate the Link Layer's services, and a Transport Provider makes the RegisterServiceProvider call to enable Transport Clients to locate the Transport Provider's services.

A Service Provider also passes configuration information in a structure called the Service Provider Descriptor Block (see Protocol Manager Status Data Structures). Each Service Provider must specify a unique name.

Procedural Interface

RegisterServiceProvider (pbProtManDeviceSpec, cbProtManDeviceSpec, prgRequestCodes, srgRequestCodes, pSPDB, sSPDB): ErcType

where

pbProtManDeviceSpec
cbProtManDeviceSpec

describe a device specification for the Protocol Manager.

prgRequestCodes
srgRequestCodes

describe an array of words: the request codes served by this Service Provider.

pSPDB
sSPDB

describe the Service Provider Descriptor Block (detailed later in the subsection Service Provider Descriptor Block).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	0C159h
12	pbProtManDeviceSpec	4	
16	cbProtManDeviceSpec	2	
18	prgRequestCodes	4	
22	srgRequestCodes	2	
24	pSPDB	4	
28	sSPDB	2	

Possible errors returned:

ErcInvalidSize, ErcNoSpace, ErcInvalidName, ErcDuplicateName.

The structure for rgRequestCodes has the following format when the Service Provider is a Link Layer:

Offset	Field	Size (Bytes)
0	OpenStationSL request code	2
2	OpenStationLL request code	2
4	CloseStation request code	2
6	ReadDLFrame request code	2
8	WriteDLFrame request code	2
10	DirectStation request code	2
12	DirectLink request code	2

The structure for rgRequestCodes has the following format when the Service Provider is a Transport Layer:

Offset	Field	Size (Bytes)
0	t_open request code	2
2	t_close request code	2
4	t_rcv/t_rcvudata/t_rcvuderr/ t_sndudata request code	2
6	t_snd request code	2
8	t_accept/t_bind/t_connect/t_rcvconnect/ t_listen/t_look/t_getstate/t_rcvdis/ t_rcvrel/t_snddis/t_sndrel/t_optmgmt/ t_unbind request code	2
10	t_sync/t_getinfo request code	2
12	request code routed by Device Specification for deinstall/status monitor	2

A Transport Layer may supply zero in any of these fields to indicate that it does not support the corresponding function. For instance, a Transport Layer which provides only a connectionless service would return zero for all requests which are only used in connection-oriented service.

Description

A Service Provider makes the DeregisterServiceProvider call to the Protocol Manager to notify it that the Service Provider will no longer serve the previously declared set of request codes. This request should be issued by the Service Provider just prior to allowing itself to be deinstalled.

Procedural Interface

DeregisterServiceProvider (pbProtManDeviceSpec, cbProtManDeviceSpec, pbServiceName, cbServiceName): ErcType

where

pbProtManDeviceSpec
cbProtManDeviceSpec

describe a device specification for the desired Protocol Manager.

pbServiceName
cbServiceName

describe the Service Name of the Service Provider which is being deregistered.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	0C15Ah
12	pbProtManDeviceSpec	4	
16	cbProtManDeviceSpec	2	
18	pbServiceName	4	
22	cbServiceName	2	

Possible errors returned:

ErcInvalidName, ErcNotRegistered, ErcInvalidUser.

Description

A Service Client makes the RegisterServiceClient call to the Protocol Manager to declare its name, its own device specification (which the Service Client should determine programmatically), and the request code it serves for its query request. This request is optional for the Service Client, as it plays no part in connecting a Client with its Service Provider.

This request is used to help manage a complex network. It allows the development of monitor utilities which keep track of the status of every software layer in a network. All Client Names should be unique in the network.

Note that a program may be both Service Providers and Service Clients. In this case the program would issue both a RegisterServiceProvider and a RegisterServiceClient.

Procedural Interface

RegisterServiceClient (pbProtManDeviceSpec, cbProtManDeviceSpec, pbClientName, cbClientName, pbClientDeviceSpec, cbClientDeviceSpec, pAuxInfoStruct, sAuxInfoStruct): ErcType

where

pbProtManDeviceSpec
cbProtManDeviceSpec

describe a device specification for the desired Protocol Manager.

pbClientName
cbClientName

describe the Service Client Name.

pbClientDeviceSpec
cbClientDeviceSpec

describe a device specification for the Service Client. A value of zero for cbClientDeviceSpec means that no Device Specification for the Service Client is supplied.

pAuxInfoStruct
sAuxInfoStruct

describe a structure, of which the only currently defined element is the Service Client's Query Request code (a word). A value of zero means that no request code is supplied.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	4
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	0C15Bh
12	reserved	6	
18	pbProtManDeviceSpec	4	
22	cbProtManDeviceSpec	2	
24	pbClientName	4	
28	cbClientName	2	
30	pbClientDeviceSpec	4	
34	cbClientDeviceSpec	2	
36	pAuxInfoStruct	4	
40	sAuxInfoStruct	2	

Possible errors returned:

ErcInvalidSize, ErcNoSpace, ErcInvalidName, ErcDuplicateName.

Description

A Service Client makes the DeregisterServiceClient call to inform the Protocol Manager that the Service Client will no longer use the services of its Service Provider(s) and will no longer serve the previously declared query request code. This request should be issued by the Service Client prior to allowing itself to be deinstalled. This request is optional for the Transport Client.

Procedural Interface

DeregisterLinkClient (pbProtManDeviceSpec, cbProtManDeviceSpec, pbClientName, cbClientName): ErcType

where

pbProtManDeviceSpec
cbProtManDeviceSpec

describe a device specification for the desired Protocol Manager.

pbClientName
cbClientName

describe the Service Client Name. This name must be unique within the network.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	0C15Ch
12	pbProtManDeviceSpec	4	
16	cbProtManDeviceSpec	2	
18	pbClientName	4	
22	cbClientName	2	

Possible errors returned:

ErcInvalidSize, ErcNotRegistered, ErcInvalidUser.

Description

A Service Client makes a RequestServiceProvider call to the Protocol Manager to determine the set of request codes served by the desired Service Provider and the Service Provider's device specification. The Service Client specifies the Service Provider name (excluding the Device Specification). The Protocol Manager interprets the Parameter Definition File (PDF) on behalf of the Service Client and returns the requested Service Provider's Device Specification for use in the subsequent request (for instance, OpenStationSL, OpenStationLL, or t_open) to the Service Provider.

Procedural Interface

RequestServiceProvider (fPDFName, pbProtManDeviceSpec, cbProtManDeviceSpec, pPDFName, sPDFName, pStructRet, sStructRet): ErcType

where

fPDFName

is a flag. If set to TRUE, the pb/cb pair for PDFName specifies a Parameter Descriptor File (PDF) name. If set to FALSE, the pb/cb pair for PDFName specifies a Parameter Descriptor String.

pbProtManDeviceSpec
cbProtManDeviceSpec

describe a device specification for the desired Protocol Manager.

pPDFName
sPDFName

describe either a Parameter Descriptor String or a file specification for a Parameter Descriptor File.

pStructRet
sStructRet

describe the structure below, which is filled in by the Protocol Manager.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	0C15Dh
12	fPDFName	1	
13	Reserved	5	
18	pbProtManDeviceSpec	4	
22	cbProtManDeviceSpec	2	
24	pPDFName	4	
28	sPDFName	2	
30	pStructRet	4	
34	sStructRet	2	

The structure for StructRet has the following format:

Offset	Field	Size (Bytes)
0	Length of Device Specification	2
2	Device Specification for Service Provider	X
2+X	Size of request code array	2
4+X	rgRequestCodes	Y
4+X+Y	Length of Parameter Return Area	2
6+X+Y	Parameter Return area	Z

The format of rgRequestCodes is the same as for RegisterServiceProvider. A Transport Client should check for request codes which are zero: these indicate that the corresponding request is not supported by the Transport Layer.

Possible errors returned:

Any file system error, ErcBadRetSize, ErcInvalidName, ErcNotRegistered, ErcInvalidPDF.

Description

The QueryProtocolManager request returns the information specified in the Service Provider Descriptor Block for all active Service Providers. The Protocol Manager always contains this information which is updated when necessary by the Service Providers using the UpdateProtocolManager request. The QueryProtocolManager request also returns the information given from Service Clients in the RegisterServiceClient request. The request codes for all of the Service Provider's "query" requests are also returned enabling the caller to query any Service Provider directly for more detailed installation and statistical information. In addition, any Service Clients which have issued a RegisterServiceClient request will have their Client Name, Device Specification, and "query" request code returned in this request. See the section on Protocol Manager Status Data Structures for the formats in which the data is returned.

Procedural Interface

QueryProtocolManager (pbProtManDeviceSpec, cbProtManDeviceSpec, pProtManStatRet, sProtManStatRet, psProtManStatRet, ssProtManStatMax): ErcType

where

pbProtManDeviceSpec
cbProtManDeviceSpec

describe a device specification for the desired Protocol Manager.

pProtManStatRet
sProtManStatRet

describe a buffer where the Status information is to be returned by the Protocol Manager.

psProtManStatRet
ssProtManStatRet

describe a word where the Protocol Manager returns the number of bytes of Status information actually returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	0C15Eh
12	pbProtManDeviceSpec	4	
16	cbProtManDeviceSpec	2	
18	pProtManStatRet	4	
22	sProtManStatMax	2	
24	psProtManStatRet	4	
28	ssProtManStatMax	2	2

Possible errors returned:

ErcBadRetSize.

The structure for ProtManStatRet has the following format:

Offset	Field	Size (Bytes)
0	Version of this copy of Protocol Manager (The high order word is the major revision level and the low order word is the minor revision level.)	4
4	Number of Service Providers registered	2
6	Number of Service Clients registered	2
8	Array of Service Provider Descriptor Blocks, (one for each Service Provider registered)	N
8+N	Array of Service Client Descriptor Blocks, (one for each Service Client registered)	M

Description

A Service Provider makes the UpdateProtocolManager call to to notify the Protocol Manager that a change or event such as a line down condition or new Service Client issuing an open request, such as an OpenStationSL, OpenStationLL, or t_open, has occurred. An updated SPDB block is passed to the Protocol Manager.

Procedural Interface

UpdateProtocolManager (pbProtManDeviceSpec, cbProtManDeviceSpec, pSPDB, sSPDB): ErcType

where

pbProtManDeviceSpec
cbProtManDeviceSpec

describe a device specification for the desired Protocol Manager.

pSPDB
sSPDB

describe the Service Provider Descriptor Block.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	0C15Fh
12	pbProtManDeviceSpec	4	
16	cbProtManDeviceSpec	2	
18	pSPDB	4	
22	sSPDB	2	

Possible errors returned:

ErcInvalidSize, ErcInvalidName, ErcNotRegistered, ErcInvalidUser.

Protocol Manager Status Data Structures

Service Provider Descriptor Block (SPDB)

This structure is passed to the Protocol Manager by RegisterServiceProvider and UpdateProtocolManager, and returned by the Protocol Manager in response to QueryProtocolManager.

Offset	Field	Size (Bytes)
0	Length of remainder of this SPDB	2
2	Size of Service Provider Name	1
3	Service Provider Name	X
3+X	Size of Service Provider Device Specification	1
4+X	Service Provider Device Specification	Y
4+X+Y	Number of Channels	1
5+X+Y	Size of Channel Names field	1
6+X+Y	Channel Name(s) [separated by a colon]	Z
6+X+Y+Z	Size of Channel Types field	1
7+X+Y+Z	Channel Type(s) [separated by a colon]	W
7+X+Y+Z+W	Number of defined Service Clients	1
8+X+Y+Z+W	Provider Status flag (up/down; down = 0)	1
9+X+Y+Z+W	Query request code (DirectLink for Link Layers) (This field only present in SPDB when SPDB is from QueryProtocolManager, not RegisterServiceProvider.)	2

The fields above which refer to "Channels" are intended for flexible use by the programmer. A Link Layer might refer to

:A: or :[COMM]A:

for Channel Names. Channel Types might refer to

:RS-232: or :X21:

A Transport Layer, in the Channel Types field, might refer to

:X25: or :Ethernet:

Service Client Descriptor Block (SCDB)

This structure is returned by the Protocol Manager in response to QueryProtocolManager.

Offset	Field	Size (Bytes)
0	Length of remainder of this SCDB	2
2	Size of Service Client Name	1
3	Service Client Name	12
15	Size of Service Client Device Specification	1
16	Service Client Device Specification	N
16+N	Service Client Query request code	2

3
Link Layer

Link Layer Overview

The CTOS Link Layer Interface is modeled on the ANSI/IEEE standard 802.2 and the ISO/DIS standard 8802.2, although not all protocols provided by the Link Layers will necessarily conform to these specifications. These specifications describe, in an abstract fashion, the services to be provided by a Data Link Layer service provider. Such a service provider is responsible for error-free delivery of data packets between adjacent nodes in a network.

The CTOS Link Layer Interface provides a standard programming interface through which software programs implementing Layer Three (i.e., OSI Network Layer, X.25 Packet layer, IP Layer, SNA Path Control) can access services provided by Layer Two software (Link Layer) without detailed knowledge of the underlying link type. Ideally, the upper layer will have no knowledge of the link layer details. Even in less-than-ideal circumstances, this standard

- Provides flexibility in configuration
- Facilitates support of new links to an upper layer
- Maximizes the benefits when new link layers are developed

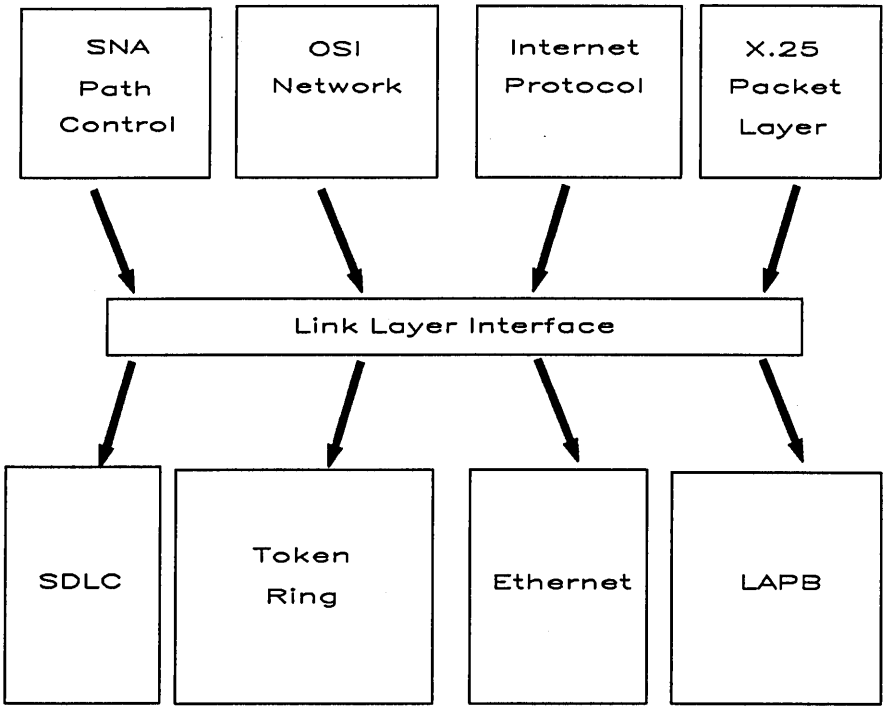


Figure 3-1. Link Layer

The CTOS/Open Link Layer Interface defines the programmatic interface between the Link Layer Service Providers (*Link Layers*) and the Layer Three implementations, or link layer service users (*Link Clients*).

A Link Layer handles one or more Link Clients. Each Link Client accesses the Link Layer through a Link Layer Service Access Point (LSAP). Some of the services provided by a Link Layer apply to the link layer as a whole, while others are specific to one LSAP. CTOS LSAP requests (*station* requests) are routed by station handle once the LSAP is opened. Link requests are routed by device specification.

There are three elements of the CTOS Link Layer Interface which facilitate device independence: the Station Descriptor File, the Link Layer Name, and the Protocol Manager.

Parameter Definition File for Link Layers

A Link Client calls either OpenStationSL or OpenStationLL to establish connection with the Link Layer. Link dependencies are removed from the parameters of these procedures by placing them in a Parameter Definition File. The parameters in the file are dependent upon the type of the Link Layer, and may include a type-of-operation indicator. Every parameter in the file is terminated by a new line character (hexadecimal 0A). Comments delimited by a pair of colons are allowed in the string. Optionally, a string itself can be directly specified in the OpenStation call; in this case, the same rules must be followed.

The only required information is the Link Layer Name (excluding the Device Specification which is supplied by the Protocol Manager), which is the first entry in the file or string. In addition, if the Link Layer allows the option to not receive Events in response to ReadDLFrame, the parameter "Events Requested?" should be the second entry. Any comments in the PDF must be deleted prior to passing the PDF string in the OpenStation call. This function will be done by the Protocol Manager if the PDF is a file.

The following example is a Parameter Definition File for an SDLC Link Layer, as used by an SNA Path Control Link Client:

```
:Link Layer Name:SDLC
:Events Requested?:Yes
:Line Address:1
:Initial connection?:Yes
:User Handles XID?:No
:Switched ID:0008D
:ID Block:03D
```

The next example is a Parameter Definition File for an Ethernet Link Layer, as used by an OSI Network Layer Link Client:

```
:Link Layer Name:Ethernet
:Events Requested?:Yes
:Local LSAP:FE
```

In either of these two cases, more parameters could be added to the PDF to accomodate implementation-dependent requirements. Product-specific parameter formats and permitted values should be supplied in the product documentation for each product. The Parameter Definition File for the Link Layer API is sometimes also called a Station Descriptor File (SDF).

Link Layer Name

A Link Layer Name is a name assigned to or by a link layer implementation at installation time. The maximum length of a name for this purpose is 12 characters. Clients of the Link Layer relate to it by the name. The combination of the Link Layer Name and Station Descriptor File provides a Link Layer (device) type independent interface.

Link Layer Services

Link Layers serve the following set of requests:

OpenStationSL
OpenStationLL
CloseStation
ReadDLFrame
WriteDLFrame
DirectStation
DirectLink

These requests are common to all Link Layers. A unique set of request codes is assigned to each Link Layer type. However, the layout and parameters for the request blocks are always the same.

All requests are available by either Request or Procedural interface. In order for Service clients to achieve device independence, only the Request interface should be used. The Procedural interface is provided as a convenience for non-generic applications which have hard-coded the procedural request names and provide a *rqLabl.asm* file for those requests.

CTOS originates the Abort, Terminate, Change User Number, and Swapping requests. Link Layers should process these as described in Chapter 8 of *CTOS/Open Programming Practices and Standards: Application Design.*

Description

The OpenStationLL request opens an LSAP for the calling Link Client. In addition, it may optionally open an initial link connection if the SDF so specifies.

A Link Client makes the OpenStationLL call to open a station after obtaining the request code set and Link Layer Name, including Device Specification, from the Protocol Manager using the RequestServiceProvider procedure. The Link Layer marks the Station Handle as long-lived.

Procedural Interface

OpenStationLL (pbLLDeviceSpec, cbLLDeviceSpec, pbClientName, cbClientName, pSDString, sSDString, pSthRet, pcbMaxDataRet): ErcType

where

pLLDeviceSpec
sLLDeviceSpec

describe a device specification for the desired Link Layer.

pClientName
sClientName

describe a string containing the Link Client's name. The maximum size allowed is 12 characters.

pSDString
sSDString

describe the SDF input string.

pSth

points to a word where the Station Handle is returned. This handle is used in subsequent requests to the Link Layer.

pcbMaxDataRet

points to a word where the maximum size of user data is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	reserved	6	
18	pbLLDeviceSpec	4	
22	cbLLDeviceSpec	2	
24	pbClientName	4	
28	cbClientName	2	
30	pSDString	4	
34	sSDString	2	
36	pSthRet	4	
40	sSthRet	2	2
42	pcbMaxDataRet	4	
46	scbMaxDataRet	2	2

The Link Layer system service will record the user number of the OpenStation request to prevent subsequent access to the station by other users until the CloseStation is issued by the opening user.

Possible errors returned:

ErcInvalidSDStringFmt, ErcInvalidSDContent, ErcAlreadyOpen, ErcNoAddress, ErcLinkDown, ErcBadLinkClientName, ErcNoResources, ErcBadLLName.

Description

The OpenStationSL request opens an LSAP for the calling Link Client. In addition, it may optionally open an initial link connection if the SDF so specifies.

A Link Client makes the OpenStationSL call to open a station after obtaining the request code set and Link Layer Name, including Device Specification, from the Protocol Manager using the RequestServiceProvider procedure. The Link Layer marks the Station Handle as short-lived.

Procedural Interface

OpenStationSL (pbLLDeviceSpec, cbLLDeviceSpec, pbClientName, cbClientName, pSDString, sSDString, pSthRet, pcbMaxDataRet): ErcType

where

pbLLDeviceSpec
cbLLDeviceSpec

describe a device specification for the desired Link Layer.

pbClientName
cbClientName

describe a string containing the Link Client's name. The maximum size allowed is 12 characters.

pSDString
sSDString

describe the SDF input string.

pSth

points to a word where the Station Handle is returned. This handle is used in subsequent requests to the Link Layer.

pcbMaxDataRet

points to a word where the maximum size of user data is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	reserved	6	
18	pbLLDeviceSpec	4	
22	cbLLDeviceSpec	2	
24	pbClientName	4	
28	cbClientName	2	
30	pSDString	4	
34	sSDString	2	
36	pSthRet	4	
40	sSthRet	2	2
42	pcbMaxDataRet	4	
46	scbMaxDataRet	2	2

The Link Layer system service will record the user number of the OpenStation request to prevent subsequent access to the station by other users until the CloseStation is issued by the opening user.

Possible errors returned:

ErcInvalidSDStringFmt, ErcInvalidSDContent, ErcAlreadyOpen, ErcNoAddress, ErcLinkDown, ErcBadLinkClientName, ErcNoResources, ErcBadLLName.

Description

The CloseStation request is used to release resources assigned by the Link Layer and make the station (LSAP) available to another user. If this request is issued while link connections are open, those connections are automatically closed.

Procedural Interface

CloseStation (Sth): ErcType

where

Sth

is the Station Handle returned by the OpenStation request.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	Sth	2	

If the user has any outstanding requests at the time of issuance of the CloseStation, all the requests will be returned before the close request is completed.

Possible errors returned:

ErcWrongUser, ErcNotOpen.

Description

The Link Client issues a ReadDLFrame to request the next received frame or a Link Layer event. The delivered field includes only the user data portion of the frame. The received message can be either connectionless (for instance, a UI frame) or connection-oriented (for instance, an I frame).

Procedural Interface

ReadDLFrame (Sth, pFrameRet, sFrameMax, psFrameRet): ErcType

where

Sth

is the Station Handle returned by the OpenStation request.

pFrameRet
sFrameMax

describe a structure where the information frame, the event code, and the remote LSAP of the frame's sender, are returned.

psFrameRet

points to a word in which the size of the returned frame is stored.

Request Block

Note that this request may be implemented as a CTOS Read/Write (%RW) request. The Link Client may issue a ReadDLFrame for up to 65535 bytes in a single request, if the Link Layer will allow that large a frame. To the Link Layer, this will appear as multiple requests.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	Sth	2	
14	Reserved	4	
18	pFrameRet	4	
22	sFrameMax	2	
24	psFrameRet	4	
28	ssFrameRet	2	2

The structure described by pFrameRet/sFrameRet has the following form:

Offset	Field	Size (Bytes)
0	EventCode	2
2	RemoteLSAPtype	2
4	RemoteLSAPLength	2
6	RemoteLSAPData	variable
6+var	FrameType	2
8+var	FrameLen	2
10+var	FrameData	variable2

If the EventCode is non-zero, the values in the remainder of the structure are undefined. See Appendix A for a list of the EventCodes and their meaning.

RemoteLSAPtype indicates the format in which the address is returned:

0	DLCi
1	IEEE LSAP format, SAP + MAC address
2	IEEE broadcast LSAP
3	DIX Ethernet
4	HDLC format (one-byte address)

The value of RemoteLSAPLength is media-dependent. Typical values are:

2	for a DLCi
8	Connectionless Token-Ring, SAP plus MAC Address or Connectionless Ethernet, SAP plus MAC Address

FrameType indicates the type of frame received (other values of FrameType may be used as a ServiceClass indicator for connectionless frames):

0	I frame (or, generally, any connection-oriented frame carrying data)
1	UI frame (or any connectionless frame carrying data)
2	TEST frame
3	XID command with Poll bit set
4	XID response with Final bit set
5	XID response without Final bit set
6	XID command without Poll bit set

Possible errors returned:

ErcWrongUser, ErcNotOpen, ErcRcvDataTrunc, ErcNullBuffer, ErcLinkDown, ErcReqCanceled, ErcStnClosed, ErcDeinstallLink, ErcNoResources, ErcReceiveTruncation, ErcInvalidFrameSpec, ErcInvalidLSAPSpec, ErcLineDown.

Description

The Link Client issues a WriteDLFrame call to request the Link Layer to transmit a message from the station. This message can be either connectionless (for instance, a UI frame) or connection-oriented (for instance, an I frame).

Procedural Interface

WriteDLFrame (Sth, pFrame, sFrame, sFrameTotal, pbCountRet, cbCountRet): ErcType

where

Sth

is the Station Handle returned by OpenStation.

pFrame
sFrame

describe a structure containing data to be transmitted and the remote LSAP to which the data is to be transmitted.

sFrameTotal

indicates whether the data pointed to by pFrame/sFrame is complete. If the frame to be transmitted by the Link Layer is too large to be sent in a single request, multiple requests can be used. *sFrameTotal* should be set to the size of the total frame. The Link Layer knows that it has received the entire frame when the sum of *sFrame* for individual requests is equal to *sFrameTotal*. If *sFrameTotal* is equal to zero, then *sFrame* is taken to indicate the length of this frame and the frame is assumed to be complete.

pbCountRet
cbCountRet

describe a word where the count of bytes successfully transmitted is returned.

Request Block

Note that this request may be implemented as a CTOS Read/Write (%RW) request. The Link Client may issue a WriteDLFrame for up to 32767 bytes in a single request, if the Link Layer will allow that large a frame. To the Link Layer, this will appear as multiple requests.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	Sth	2	
14	Reserved	2	
16	sFrameTotal	2	
18	pFrame	4	
22	sFrame	2	
24	pbCountRet	4	
28	cbCountRet	2	2

The structure described by pFrame/sFrame has the following form (on the first request only if the frame is to be conveyed in a series of requests; on the remainder this pb/cb points to FrameData):

Offset	Field	Size (Bytes)
0	RemoteLSAPType	2
2	RemoteLSAPLength	2
4	RemoteLSAPData	variable
4+var	FrameType	2
6+var	FrameLength	2
8+var	FrameData	variable2

RemoteLSAPtype indicates the format in which the address is passed. A value of zero indicates that RemoteLSAPData contains a DLCi. The value of RemoteLSAPLength is media-dependent. Typical values are 2 (for a DLCi), 8 (Connectionless Token-Ring, SAP plus MAC Address), or 10 (Connectionless Ethernet, SAP plus MAC Address).

FrameType indicates the type of frame received (other values of FrameType may be used as a ServiceClass indicator for connectionless frames):

0	I frame (or, generally, any connection-oriented frame carrying data)
1	UI frame (or any connectionless frame carrying data)
2	TEST frame
3	XID command with Poll bit set
4	XID response with Final bit set
5	XID response without Final bit set
6	XID command without Poll bit set

Possible errors returned:

ErcWrongUser, ErcNotOpen, ErcXmtDataTrunc, ErcNullBuffer, ErcLinkDown, ErcReqCanceled, ErcStnClosed, ErcDeinstallLink, ErcInvalidState, ErcLineDown, ErcLinkReset, ErcNoResources, ErcInvalidDataSize, ErcInvalidCommand.

Description

The Link Client issues a DirectStation request to direct the station operation of the Link Layer. The DirectStation request is routed by FileHandle and so can be issued only by the user that issued the OpenStation. An error code ErcWrongUser is returned by the Link Layer if this request is received from a user other than the one that issued the OpenStation. Conceptually, this request is used by the Link Client only to request any service which is specific to an LSAP or to a Link Connection. See the section on DirectStation commands for specifics of each *bCommand* option.

Procedural Interface

DirectStation (Sth, bCommand, AuxByte, AuxWord, pSend, sSend, pReceive1, sReceive1, pReceive2, sReceive2): ErcType

where

Sth

is the Station Handle returned by OpenStation.

bCommand

defines the supported Link Layer commands. Values ranging from 0-127 indicate commands common to all Link Layers, and values ranging from 128-255 indicate Link Layer dependent commands. See the section on DirectStation commands for specific value definitions.

AuxByte
AuxWord

are auxiliary fields that have values dependent upon the command entered in the *bCommand* parameter.

pSend
sSend

describe the data buffer to be sent to the Link Layer.

pReceive1
sReceive1

describe a buffer where the Link Layer is to return data.

pReceive2
sReceive2

describe an area (normally a word) where the Link Layer is to return data (normally the actual size of the data returned into pReceive1/sReceive1).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	Sth	2	
14	bCommand	1	
15	AuxByte	1	
16	AuxWord	2	
18	pSend	4	
22	sSend	2	
24	pReceive1	4	
28	sReceive1	2	
30	pReceive2	4	
34	sReceive2	2	

Possible errors returned:

ErcBadCommand, ErcNotSupported, ErcLinkDown, ErcWrongUser, ErcNotOpen, ErcLineDown, ErcReqCanceled, ErcStnClosed, ErcBufferTooSmall.

Description

Any program can issue a DirectLink request to direct the Link Layer operation. The DirectLink request is routed by FileSpec where the DirectStation request is routed by FileHandle. Conceptually, this request is used by the Link Client, or by any other program, to request a service which must apply to the entire Link Layer.

Procedural Interface

DirectLink (bCommand, pLLDeviceSpec, sLLDeviceSpec, pSend, sSend, pReceive1, sReceive1, pReceive2, sReceive2): ErcType

where

bCommand

defines the supported Link Layer commands. Values ranging from 0-127 indicate commands common to all Link Layers, and values ranging from 128-255 indicate Link Layer dependent commands. See the section on DirectLink commands for specific value definitions.

pLLDeviceSpec
sLLDeviceSpec

describe a device specification for the desired Link Layer. The Link Name must also be specified here following the device specification.

pSend
sSend

describe the data buffer to be sent to the Link Layer.

pReceive1
sReceive1

describe a buffer where the Link Layer is to return data.

pReceive2
sReceive2

describe an area (normally a word) where the Link Layer is to return data (normally the actual size of the data returned into pReceive1/sReceive1).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	bCommand	1	
13	Reserved	5	
18	pLLDeviceSpec	4	
22	sLLDeviceSpec	2	
24	pSend1	4	
28	sSend1	2	
30	pReceive1	4	
34	sReceive1	2	
36	pReceive2	4	
40	sReceive2	2	

Possible errors returned:

ErcBadCommand, ErcNotSupported, ErcLLActive, ErcBadLLName, ErcBufferTooSmall.

See the section on DirectLink commands for specifics of each *bCommand* option.

Link Layer Status Data Structures

Generic Statistical Header Block (GHB)

The Generic Statistical Header Block (GHB) is returned by the Link Layer as a response to a DirectStation (with QueryStation command) or for a DirectLink (with QueryLink) command. The fields on this page are completely generic. All Link Layers must support these. The remainder of the fields, on the next page, are a recommended ordering of statistics with general applicability.

Offset	Field	Size (Bytes)
0	Size of Link Layer Name	1
1	Link Layer Name	X
1+X	Size of Link Layer Device Specification	1
2+X	Link Layer Device Specification	Y
2+X+Y	Number of Channels	1
3+X+Y	Size of Channel Names field	1
4+X+Y	Channel Name(s) [separated by a colon]	Z
4+X+Y+Z	Size of Channel Types field	1
5+X+Y+Z	Channel Type(s) [separated by a colon]	W
5+X+Y+Z+W	Number of defined Link Clients	1
6+X+Y+Z+W	Line Status flag (Physical Layer up/down)	1
7+X+Y+Z+W	DirectLink request code (This field is zero when the LDB is returned by QueryLink)	2
9+X+Y+Z+W	Total Info Frames Received	4
13+X+Y+Z+W	Total Info Frames Transmitted	4
29+X+Y+Z	Switched Block ID	4
33+X+Y+Z	# of TEST commands received	2
35+X+Y+Z	# of TEST commands transmitted	2
37+X+Y+Z	Validity mask (nSummaryMask)	1
38+X+Y+Z	# of machine checks	2
40+X+Y+Z	# of communications checks	2
42+X+Y+Z	# of program checks	2
44+X+Y+Z	Validity mask (nCommMask1)	1
45+X+Y+Z	Validity mask (nCommMask2)	1

Offset	Field	Size (Bytes)
46+X+Y+Z	# of nonproductive timeouts	1
47+X+Y+Z	# of idle timeouts	1
48+X+Y+Z	# of retransmissions	1
49+X+Y+Z	# of receiver overruns	1
50+X+Y+Z	# of transmitter underruns	1
51+X+Y+Z	# of conection problems	1
52+X+Y+Z	# of FCS errors	1
53+X+Y+Z	# of primary aborts	1
54+X+Y+Z	# of frame reject responses transmitted	1
55+X+Y+Z	# of DCE errors	1
56+X+Y+Z	# of transmit timeouts	1
57+X+Y+Z	# of invalid adapter status	1
58+X+Y+Z	# of adapter machine checks	1
62+X+Y+Z	# Total Frames Received	4
66+X+Y+Z	# Total Frames Transmitted	4
70+X+Y+Z	# Total Error Frames	4
74+X+Y+Z	cb Memory allocated internally in Link Layer	4
78+X+Y+Z	cb Memory free internally in Link Layer	4
82+X+Y+Z	Size of following additional information	2

Any additional statistics returned should include Link Layer specific information. One example would be for an Ethernet Link Layer to return the number of collisions (data which does not apply to other Link Layers).

DirectStation Commands

Link Layer Independent Commands

All Link Layers should support the independent commands listed in the following table. If, however, a command is not supported by the Link Layer, an error code indicating non-support is returned (ErcNotSupported, 31105). These commands include standard ISO 8802.2 and IEEE 802.2 functions and management functions common to all Link Layers.

bCommand Value	Command Name	Description
1	Query station	Requests performance statistics for a station.
2	Reset statistics data	Resets statistics counters to zero.
3	Cancel requests	Causes all outstanding requests to be returned.
4	Reset Logical Link	Causes the Link Layer to reinitialize the specified connection. The AuxWord parameter should be the DLCi.
5	Open Logical Link	Requests the Link Layer to initiate an LLC connection with a remote RLSAP/RMAC address.
6	Close Logical Link	Requests the Link Layer to terminate an LLC connection; requests disconnection. The AuxWord parameter should be the DLCi.
10	Set busy mode	Causes the Link Layer to mark this station or connection as "busy." For instance, an HDLC Link Layer would start sending RNR. The Link Layer should also enter busy mode whenever it does not have an outstanding read request from the Link Client. The AuxWord parameter should be the DLCi.

bCommand Value	Command Name	Description
11	Set not busy mode	Causes the Link Layer to mark this station or connection as "not busy." For instance, an HDLC Link Layer would resume sending RR. The Link Layer should also exit busy mode whenever it receives a read request from a Link Client who did not have one outstanding. The AuxWord parameter should be the DLCi.

For all commands except Query Station, Open Logical Link, and Close Logical Link, all of the request block parameters except AuxWord are not used and should be set to zero.

Query Station Command

When Query Station (1) is specified in the *bCommand* parameter, specify the last parameters for the DirectStation request as follows:

- *pSend*: Not used.
- *sSend*: Not used.
- *pReceive1*: A pointer to a buffer where the GHB is returned.
- *sReceive1*: The size of the GHB.
- *pReceive2*: A pointer to a word where the Link Layer returns the size of the GHB.
- *sReceive2*: 2.

Open Logical Link Command

When Open Logical Link (5) is specified in the *bCommand* parameter, specify the last parameters for the DirectStation request as follows:

- *AuxByte*: Class of Service. When multiple priorities are available, this selects which should be used for this connection.
- *pSend*: A pointer to a Remote LSAP structure.
- *sSend*: The size of the structure.
- *pReceive1*: A pointer to a word where the Data Link Connection indicator (DLCi) is returned.
- *sReceive1*: 2.
- *pReceive2*: Not used.
- *sReceive2*: Not used.

The structure for the Remote LSAP has the following format:

Offset	Field	Size (Bytes)
0	RemoteLSAPType	2
2	RemoteLSAPLength	2
4	RemoteLSAPData	variable
4+variable	Optional Data	variable

The DLCi is the Data Link Connection indicator for the connection. The confirmation status is returned in ercRet of the request block.

Close Logical Link Command

The Link Client issues a Close Logical Link command to terminate a connection with a remote SAP address. When the Link Layer responds to the request, an ercRet of zero confirms that the disconnect was successful.

When Close Logical Link (6) is specified in the *bCommand* parameter, specify the last parameters for the DirectStation request as follows:

- *AuxWord*: The DLCi returned by Open Logical Link command.
- *pSend*: Not used.
- *sSend*: Not used.
- *pReceive1*: Not used.
- *sReceive1*: Not used.
- *pReceive2*: Not used.
- *sReceive2*: Not used.

Link Layer Dependent Commands

Commands listed in the following table may or may not be supported by a Link Layer. If a command is not supported by the Link Layer, an error code indicating non-support is returned (ErcNotSupported, 31105). These commands are intended to allow for hardware or device dependencies which cannot be avoided.

bCommand Value	Command Name	Description
128	Transmit XID	Causes an SDLC/SNA XID to be transmitted.
131	Transmit TEST	Causes a TEST frame to be transmitted.
133	Notify Link Role	If a Link Layer can handle different roles (i.e., primary vs. secondary), this command instructs the Link Layer which role to assume.
134	Connect Logical Link	Confirms the initiation of a Token-Ring LLC connection and supplies Token-Ring bridge routing information.
137	Change Timers	Dynamically change per-connection timers.
138	Change Windows	Dynamically change per-connection transmit or receive window size.
139	ISDN Mux	LAPD
140	MF-EstablishLink	LAPD
141	MF-ReleaseLink	LAPD

For the Transmit XID, Transmit Set Link Mode, Transmit DISC, Transmit TEST, the pSend/sSend parameters point to a transmit buffer and the other parameters are not used. For the Link Reset command, no parameters are used. For the Notify Link Role command, pSend/sSend point to a word indicating the role and the other parameters are not used. For the MF-EstablishLink and MF-ReleaseLink commands, AuxWord should be the DLCi. For all commands, pointers not used should be set to zero.

Connect Logical Link Command

The Link Client issues an Connect Logical Link command to complete a connection with a remote SAP address.

When Connect Logical Link (134) is specified in the *bCommand* parameter, specify the last parameters for the DirectStation request as follows:

- *AuxWord*: The DLCi returned by the Open Logical Link command.

DirectLink Commands

Link Layer Independent Commands

The commands in the following table include management functions common to all Link Layers and physical layer operations of wide applicability to many Link Layer types. If a command is not supported by a Link Layer, an error code indicating non-support is returned (ErcNotSupported).

bCommand Value	Command Name	Description
1	Query Link Layer statistics	p/sReceive1 point to a buffer where a GHB is returned.
2	Deinstall	p/sReceive1 point to a partition handle.
3	Make outgoing call	p/sSend1 point to a phone number or a X.21 PDN destination address.
4	Enable Incoming call	No parameters are used.
6	Disconnect call	No parameters are used.
7	Disable Incoming call	No parameters are used.
10	Reset Link Layer	Causes the Link Layer to reinitialize itself.

Link Layer Dependent Commands

The commands in the following table generally apply to specific Link Layers only. If the command is not supported, the Link Layer returns an error code indicating non-support (ErcNotSupported).

bCommand Value	Command Name
150	Configure
151	Individual Address SetUp
152	Add Multi-Cast or Group Address
153	Remove Multi-Cast or Group Address
154	Reset Link Layer Statistics
155	Set Functional Address
156	Reconfigure (pSend1/sSend1 point to reconfiguration data)
157	Loopback control (Local vs. Remote loopback)
158	Activate ISDN (pSend1/sSend1 points to a channel name)
159	Deactivate ISDN (pSend1/sSend1 points to a channel name)

4

Transport Layer

Transport Layer Overview

The CTOS Transport Layer Interface is modeled on the X/Open Company Ltd.'s X/Open Transport Interface. Both a library procedural interface (object module procedures) and a CTOS request interface are defined. The library procedural interface is completely identical to the X/Open Transport Interface (XTI) and is provided because it is widely recognized as an industry standard. The library procedural interface is the appropriate one to use when writing an application which must be portable between several different operating systems. The CTOS request interface is provided for performance and efficiency (especially when handling full-duplex flow). Its use is appropriate when writing a CTOS upper layer program which must be able to operate over many different transport providers.

The X/Open specification describes an API for a set of services modeled on the ISO OSI Transport layer. An OSI Transport layer service provider is responsible for error-free delivery of data between any two nodes in a network, regardless of the topology of the network or the number of nodes in-between. The X/Open specification can be used with any underlying transport protocol which delivers this service.

The CTOS Transport Layer Interface provides a standard programming interface through which software programs can access communication services provided by transport layer software (for instance, OSI Transport Layer or Transmission Control Protocol) without detailed knowledge of the underlying transport type. One example is an implementation of the OSI Session Layer using this interface to access the services of an implementation of the OSI Transport Layer. Ideally, the upper layer will have no knowledge of the transport layer details. Even in less-than-ideal circumstances, this standard

Provides flexibility in configuration

Facilitates support for new transport layers to an upper layer

Maximizes the benefits when new transport layers are developed

The transport layer described in Figure 4-1 is important because it is the lowest layer in the OSI Reference Model that provides the basic service of reliable, end-to-end data transfer needed by applications and higher layer protocols. In doing so, this layer hides the topology and characteristics of the underlying network from its users. More important, however, the transport layer defines a set of services common to layers of many contemporary protocol suites, including the International Standards Organization (ISO) protocols, the Transmission Control Protocol and Internet Protocol (TCP/IP) of the ARPANET, Xerox Network Systems (XNS), and the IBM-defined Netbeui protocol commonly used by personal computers on a local-area network.

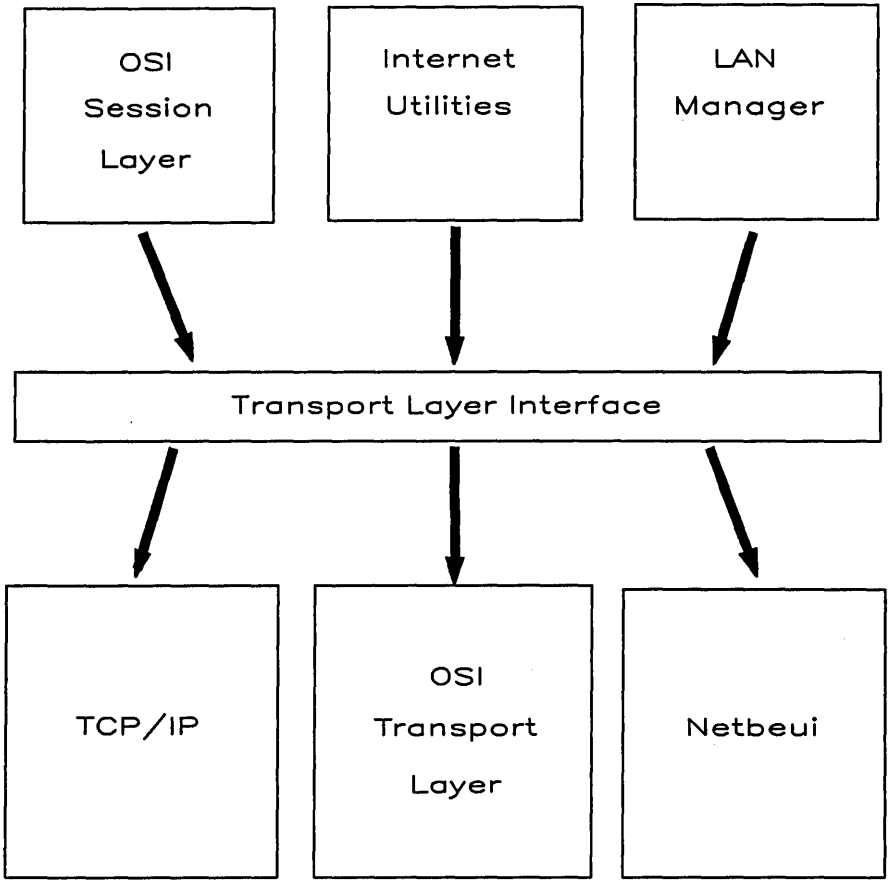


Figure 4-1. Transport Layer

X/Open defined the XTI specification as a transport service interface which is independent of any specific transport provider. The interface is provided by way of a set of library functions, originally for the C programming language (for CTOS, any language may be used). XTI was originally intended to run on various versions of UNIX.

X/Open intended XTI to describe a wide set of functions and facilities which vary in importance and/or usefulness. An application will be portable across systems incorporating XTI (for instance, between UNIX systems and CTOS) if one or both of the following is true:

- The application has been written such that it can modify its behavior according to any subset of these functions and facilities which may be supported by each of the transport providers over which the applications is intended to work, or

- The application uses only the mandatory functions which are guaranteed to be supported by the transport provider.

Note that the choice of functions and facilities may be confusing for implementors of both applications and providers. Therefore XTI distinguishes between those functions and facilities which are considered to be mandatory, and those which are optional, for the purpose of identifying the minimum workable subset. The CTOS/Open Transport Layer Interface extends the set of functions and facilities which are considered to be mandatory beyond the set considered mandatory by X/Open.

XTI was originally concerned primarily with the ISO **[ref 1]**, OSI Transport Service Definition (Connection-oriented or Connectionless) **[ref 2]**. However, it may be adapted for use over other types of transport provider. In particular, XTI has been extended to include TCP **[ref 3]** and UDP **[ref 4]** because of the popularity of these protocols.

References

[ref 1] The OSI model is described in:

ISO 7498 Information Processing Systems
Open Systems Interconnection
Basic Reference Model {IS : 1984}

[ref 2] The reference documents for ISO transport are summarized in this array:

	Connection-oriented	Connectionless
protocol definition	IS 8073-1986	IS 8602
service definition	IS 8072-1986	IS 8072/Add.1-1986

[ref 3] The reference document for TCP protocol is:

TCP Transmission Control Protocol
Military Standard
(Mil-std-1778 Source A) and RFC 793 (Source B)

[ref 4] The reference document for UDP protocol is:

UDP User Datagram Protocol
RFC 768 (Source B)

where:

Source A Defense Communication Agency
DDN Protocol Handbook (Volume One)
DOD Military Standard Protocols (DEC 1985)

Source B Defense Communication Agency
DDN Protocol Handbook (Volume Two)
DARPA Internet Protocols (Dec 1985)

Explanatory Notes

Transport Endpoints

A *transport endpoint* specifies a communication path between a transport user and a specific transport provider, which is identified by a local file descriptor (*fd*). When a user opens a transport provider identifier, a local file descriptor *fd* is returned which identifies the transport endpoint. A transport provider is defined to be the transport protocol that provides the services of the transport layer. All requests to the transport provider must pass through a transport endpoint. The file descriptor *fd* is returned by the function *t_open*() and is used as an argument to the subsequent functions to identify the transport endpoint. A transport endpoint (*fd* and local address) can support only one established transport connection at a time.

To be active, a transport endpoint must have a transport address associated with it by the *t_bind*() function. A transport connection is characterized by the association, of two active endpoints, made by using the functions of establishment of transport connection. The *fd* is a communication path to a transport provider. There is no direct assignation of the processes to the transport provider. So multiple processes which share the same *fd* (whether created by the CTOS *CreateProcess*() or *LoadTask*() operations or the POSIX *fork*() operation) may access a given communication path.

Note that to guarantee portability, the applications may only perform operations defined in the XTI on *fd*s returned by *t_open*(). Other operations are permitted but these will have system-dependent results.

Transport Providers

The transport layer may comprise one or more *transport providers* at the same time. The identifier parameter of the transport provider passed to the *t_open*() function determines the required transport provider. To keep the applications portable, the identifier parameter of the transport provider should not be hard-coded into the application source code.

An application which wants to manage multiple transport providers must call *t_open*() for each provider. For example, a server application which is waiting for incoming connect indications from several transport providers must open a transport endpoint for each provider and listen for connect indications on each of the associated file descriptors.

A transport provider may provide only connectionless service, only connection-oriented service, or both.

Association of a Process to an Endpoint

One process can simultaneously open several *fd*s. However, in synchronous mode, the process must manage the different actions of the associated transport connections sequentially. Conversely, several processes can share the same *fd* but they have to synchronize themselves so as not to issue a function that is unsuitable to the current state of the transport endpoint.

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

A process can listen for an incoming connect indication on one *fd* and accept the connection on a different *fd* which has been bound with the *qlen* parameter (see *t_bind*()) set to zero. This facilitates the writing of a listener application whereby the listener waits for all incoming connect indications on a given Transport Service Access Point (TSAP). The listener will accept the connection on a new *fd* and service the request without blocking other incoming connect indications.

Use of Same Protocol Address

If several endpoints are bound to the same protocol address, only one at the time may be listening for incoming connections. However, others may be in data transfer state or establishing a transport connection as initiators.

Modes of Services

The transport service interface supports two modes of service: *connection mode* and *connectionless mode*. A single transport endpoint may not support both modes of service simultaneously.

The connection-mode transport service is circuit-oriented and enables data to be transferred over an established connection in a reliable, sequential manner. This service enables the negotiation of the parameters and options that govern the transfer of data. It provides an identification mechanism that avoids the overhead of address transmission and resolution during the data transfer phase. It also provides a context in which successive units of data, transferred between peer users, are logically related. This service is attractive to applications that require relatively long lived, datastream-oriented interactions.

In contrast, the connectionless-mode transport service 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. No dynamic negotiation of parameters and options is supported by this service. All the information required to deliver a unit of data (for instance, destination address) is presented to the transport provider, together with the data to be transmitted, in a single service access which need not relate to any other service access. Also, each unit of data transmitted is entirely self-contained, and can be independently routed by the transport provider. This service is attractive to applications that involve short-term request/response interactions exhibit a high level of redundancy, are dynamically reconfigurable, or do not require guaranteed, in-sequence delivery of data.

Error Handling

Two levels of error are defined for the transport interface. The first is the library error level. Each library function has one or more error returns. Failures are indicated by a return value of -1. An external integer, *t_errno* (which is defined in the header **<xti.h>** for the C programming language) holds the specific error number when such a failure occurs. This value is set when errors occur but is not cleared on successful library calls, so it should be tested only after an error has been indicated. If implemented, the optional diagnostic function, *t_error*(), prints out information on the current transport error. The state of the transport provider may change if a transport error occurs.

The second level of error is the operating system service routine level. A special library level error number has been defined called [TSYSERR] which is generated by each library function when an operating system service routine fails or some general error occurs. When a function sets *t_errno* to [TSYSERR], the specific system error may be accessed through the external variable *errno*.

For example, a system error can be generated by the transport provider when a protocol error has occurred. If the error is severe, it may cause the file descriptor and transport endpoint to be unusable. To continue in this case, all users of the *fd* must close it. Then the transport endpoint may be re-opened and initialized.

Synchronous and Asynchronous Execution Modes

The transport service interface is inherently asynchronous; various events may occur which are independent of the actions of a transport user. For example, a user may be sending data over a transport connection when an asynchronous disconnect indication arrives. The user must somehow be informed that the connection has been broken.

The transport service interface supports two execution modes for handling asynchronous events, synchronous mode and asynchronous mode. In the synchronous mode of operation, the transport primitives wait for specific events before returning control to the user. While waiting, the user cannot perform other tasks. For example, a function that attempts to receive data in synchronous mode will wait until data arrives before returning control to the user. Synchronous mode is the default mode of execution. It is useful for user processes that want to wait for events to occur, or for user processes that maintain only a single transport connection.

The asynchronous mode of operation, on the other hand, provides a mechanism for notifying a user of some event without forcing the user to wait for the event. The handling of networking events in an asynchronous manner is seen as a desirable capability of the transport interface. This would enable users to perform useful work while expecting a particular event. For example, a function that attempts to receive data in asynchronous mode will return control to the user immediately if no data is available. The user may then periodically poll for incoming data until it arrives. The asynchronous mode is intended for those applications that expect long delays between events and have other tasks that they can perform in the meantime or handle multiple connections concurrently.

The two execution modes are not provided through separate interfaces or different functions. Instead, functions that process incoming events have two modes of operation: synchronous and asynchronous. The desired mode is specified through the O_NONBLOCK flag, which may be set when the transport provider is initially opened (UNIX systems allow this flag to be set using the *fcntl*() operating system service routine before any specific function or group of functions is executed, but CTOS allows this flag to be set only on *t_open*()). The effect of this flag is local to this process and is completely specified in the description of each function.

Nine (only eight if the orderly release is not supported) asynchronous events are defined in the transport service interface to cover both connection mode and connectionless-mode service. They are represented as separate bits in a bit-mask using the following defined symbolic names:

- T_LISTEN
- T_CONNECT
- T_DATA
- T_EXDATA
- T_DISCONNECT
- T_ORDREL
- T_UDERR
- T_GODATA
- T_GOEXDATA

These are described in the section entitled Event Management.

A process that issues functions in synchronous mode must still be able to recognize certain asynchronous events and act on them if necessary. This is handled through a special transport error [TLOOK] which is returned by a function when an asynchronous event occurs. The *t_look*() function is then started to identify the specific event that has occurred when this error is returned.

Another means to notify a process that an asynchronous event has occurred is polling. The polling capability enables processes to do useful work and periodically poll for one of the above asynchronous events. This facility is provided by setting O_NONBLOCK for the appropriate primitive(s).

Events and t_look()

All events that occur at a transport endpoint are stored by XTI. These events are retrievable one at a time using the *t_look*() function. If multiple events occur, it is implementation-dependent in what order *t_look*() will return the events. An event is outstanding on a transport endpoint until it is consumed. Every event has a corresponding consuming function which handles the event and clears it. Two events, T_GODATA and T_GOEXDATA are also cleared as they are returned by *t_look*(). The following table summarizes this:

Event	Cleared on *t_look*()?	Consuming XTI functions
T_LISTEN	No	*t_listen*()
T_CONNECT	No	*t_rcvconnect*()
T_DATA	No	*t_rcv{udata}*()
T_EXDATA	No	*t_rcv*()
T_DISCONNECT	No	*t_rcvdis*()
T_UDERR	No	*t_rcvuderr*()
T_ORDREL	No	*t_rcvrel*()
T_GODATA	Yes	*t_snd{udata}*()
T_GOEXDATA	Yes	*t_snd*()

Event Management

Each XTI call deals with one transport endpoint at a time. It is not possible for client programs, using the XTI library calls, to wait for several events from different sources, particularly from several transport connections at a time. Although X/Open recognizes the need for this functionality, no mechanism to support this functionality has been standardized by X/Open.

On CTOS, this function is provided by permitting the transport client to keep multiple *t_look*() requests outstanding simultaneously. (This fulfills the mechanism which X/Open calls Event Management or EM). Processes can be notified of the following events:

- T_LISTEN:

  A connect request from a remote user was received by a transport provider (connection-mode service only); this event may occur under the following conditions:

  1. file descriptor is bound to a valid address.

  2. no transport connection is established at this time.

- T_CONNECT:

  In connection mode only; a connect response was received by the transport provider; occurs after a *t_connect*() has been issued.

- T_DATA

  Normal data (whole or part of Transport Service Data Unit (TSDU)) was received by the transport provider.

- T_EXDATA

  Expedited data was received by the transport provider.

- T_DISCONNECT

  In connection mode only; a disconnect request was received by the transport provider. It may be reported on both data transfer functions and on the *t_accept*() and *t_snddis*() functions.

- T_ORDREL

  An orderly release request was received by a transport provider (connection mode with orderly release only).

- T_UDERR

  In connectionless mode only; an error was found in a previously sent datagram. It may be notified on the *t_rcvudata*() or *t_unbind*() function calls.

- T_GODATA

  Flow control restrictions on normal data flow have been lifted. Normal data may be sent again.

- T_GOEXDATA

  Flow control restrictions on expedited data flow have been lifted. Expedited data may be sent again.

XTI Overview

Overview of Connection-Oriented Mode

The connection-mode transport service consists of four phases of communication:

- Initialization/Deinitialization.
- Connection Establishment.
- Data Transfer.
- Connection Release.

A state machine is described in the subsection Transport Layer Interface Sequence of Functions which defines the legal sequence in which functions from each phase may be issued.

In order to establish a transport connection, a user (application) must:

1. Supply a *transport provider identifier* for the appropriate type of transport provider (using *t_open*()); this establishes a transport endpoint through which the user may communicate with the provider.
2. Associate (bind) an address with this endpoint (using *t_bind*()).
3. Use the appropriate connection functions (using *t_connect*(), or *t_listen*() and *t_accept*()) to establish a transport connection. The set of functions depends on whether the user is an initiator or responder.

4. Once the connection is established, normal, and if authorized, expedited data can be exchanged. Of course, expedited data may be exchanged only if:

   - The provider supports it.
   - Its use is not precluded by the selection of protocol characteristics, for instance, the use of Class 0.
   - Negotiation as to its use has been agreed between the two peer transport providers.

5. The transport connection can be released at any time by using the disconnect functions. Then the user can either de-initialize the transport endpoint by closing the file descriptor returned by *t_open*() (thereby freeing the resource for future use), or specify a new local address (after the old one has been unbound), or reuse the same address and establish a new transport connection.

Initialization/Deinitialization Phase

The functions that support initialization/deinitialization tasks are described below. All such functions provide local management functions; no information is sent over the network.

t_open() This function creates a transport endpoint and returns protocol-specific information associated with that endpoint. It also returns a file descriptor that serves as the local identifier of the endpoint.

t_bind() This function associates a protocol address with a given transport endpoint, thereby activating the endpoint. It also directs the transport provider to begin accepting connect indications if so desired.

t_optmgmt() (OPTIONAL) This function enables the user to get or negotiate protocol options with the transport provider.

t_unbind() This function disables a transport endpoint such that no further request destined for the given endpoint will be accepted by the transport provider.

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

The following functions are also local management functions, but can be issued during any phase of communication:

t_getinfo() (OPTIONAL) This function returns protocol-specific information associated with the specified transport endpoint.

t_getstate() (OPTIONAL) This function returns the current state of the transport endpoint.

t_sync() This function synchronizes the data structures managed by the transport library with the transport provider.

t_look() This function returns the current event(s) associated with the given transport endpoint.

Overview of Connection Establishment

This phase enables two transport users to establish a transport connection between them. In the connection establishment scenario, one user is considered active and initiates the conversation, while the second user is passive and waits for a transport user to request a connection.

In connection mode:

- First, the user has to establish an endpoint, or in other words to open a communications path between the application and the transport provider.
- Once established, an endpoint must be bound to an address and more than one endpoint may be bound to the same address.
- An endpoint can be associated with one, and only one, established transport connection.

- It is possible to use an endpoint to receive and enqueue incoming connect indications (only if the provider is able to accept more than one outstanding connect indication; this mode of operation is declared at the time of calling *t_bind*() by setting *qlen* greater than 0). However, if more than one endpoint is bound to the same address, only one of them may be used in this way.

- The *t_listen*() function is used to look for an enqueued connect indication; if it finds one (at the head of the queue), it returns details of the connect indication, and a local sequence number which uniquely identifies this indication or it may return a negative value with *t_errno* set to [TNODATA]. The number of outstanding connect requests to dequeue is limited by the value of the *qlen* parameter accepted by the transport provider on the *t_bind*() call.

- If the endpoint has more than one connect indication enqueued, the user should dequeue all connect indications (and disconnect indications) before accepting or rejecting any or all of them. The number of outstanding connect indications is limited by the value of the *qlen* parameter accepted by the transport provider on the call to *t_bind*().

- When accepting a connect indication, the transport service user may issue the accept on the same (listening) endpoint or on a different endpoint.

  If the same endpoint is used, the listening endpoint can no longer be used to receive and enqueue incoming connect indication. The bound protocol address will be found to be busy for the duration of the active transport endpoint. No other transport endpoints may be bound to the same protocol address while the listening endpoint is in the data transfer or disconnect phase (that is, until a *t_unbind*() call is issued).

  If a different endpoint is used, the listening endpoint can continue to receive and enqueue incoming connect requests.

- If the user issues a *t_connect()* on a listening endpoint, again, that endpoint can no longer be used to receive and enqueue incoming connect requests.

The functions that support these operations of connection establishment are:

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

t_rcvconnect() This function enables an active transport user to determine the status of a previously sent connect request. If the request was accepted, the connection establishment phase will be complete on return from this function. This function is used in conjunction with *t_connect*() to establish a connection in an asynchronous manner.

t_listen() This function enables the passive transport user to receive connect indications from other transport users.

t_accept() This function is issued by the passive user to accept a particular connect request after an indication has been received.

Overview of Data Transfer

Once a transport connection has been established between two users, data may be transferred back and forth over the connection in a full duplex way. Two functions have been defined to support data transfer in connection mode as follows:

t_snd() This function enables transport users to send either normal or expedited data over a transport connection.

t_rcv() This function enables transport users to receive either normal or expedited data over a transport connection.

In data transfer phase, the occurence of the T_DISCONNECT event implies an unsuccessful return from the called function (*t_snd*() or *t_rcv*()) with *t_errno* set to [TLOOK]. The user must then issue a *t_look*() call to get more details.

Receiving Data

If data (normal or expedited) is immediately available, then a call to *t_rcv*() returns data. If the transport connection no longer exists, then the call returns immediately, indicating failure. If data is not immediately available and the transport connection still exists, then the results of a call to *t_rcv*() depends on the mode:

- asynchronous mode:

  The call returns immediately, indicating failure. The user must continue to "poll" for incoming data, either by issuing repeated calls to *t_rcv*(), or by using the *t_look*().

- synchronous mode:

  The call is blocked until one of the following conditions becomes true:

  - data (normal or expedited) is received.
  - a disconnect indication is received.
  - a signal has arrived.

  The user may issue a *t_look*() to determine if data is available.

If a normal TSDU is to be received in multiple *t_rcv*() calls, then its delivery may be interrupted at any time by the arrival of expedited data. The application can detect this by checking the *flags* field on return from a call to *t_rcv*(); this will be indicated by *t_rcv*() returning:

- Data with T_EXPEDITED flag not set and T_MORE set: (this is a fragment of normal data).
- Data with T_EXPEDITED set (and T_MORE set or unset); this is an expedited message (whole or part of, depending on the setting of T_MORE). The provider will continue to return the expedited data (on this and subsequent calls to *t_rcv*() until the end of the Expedited Transport Service Data Unit (ETSDU) is reached, at which time it will continue to return normal data. It is the user's responsibility to remember that the receipt of normal data has been interrupted in this way.

If the data can be accepted immediately by the provider, then it is accepted, and the call returns the number of octets accepted. If the data cannot be accepted because of a permanent failure condition (for instance, transport connection lost), then the call returns immediately, indicating failure. If the data cannot be accepted immediately because of a transient condition (for instance, lack of buffers, flow control in effect), then the result of a call to *t_snd*() depends on the execution mode:

- asynchronous mode:

  The call returns immediately, indicating failure. If the failure was due to flow control restrictions, then 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 number of octets requested to be sent. The user may either retry the call to *t_snd*() with the remaining data and the T_MORE flag set appropriately or first receive notification of the clearance of the flow control restriction using *t_look*(), then resend the data.

- synchronous mode:

  The call is blocked until one of the following conditions becomes true:

  - The flow control restrictions are cleared and the transport provider is able to accept a new data unit. The *t_snd*() function then returns successfully.

  - A disconnect indication is received. In this case the *t_snd*() function returns unsuccessfully with *t_errno* set to [TLOOK]. The user can issue a *t_look*() function to determine the cause of the error. For this particular case *t_look*() will return a T_DISCONNECT event. Data that was being sent will be lost.

  - An internal problem occurs. In this case the *t_snd*() function returns unsuccessfully with *t_errno* set to [TSYSERR]. Data that was being sent will be lost.

Normal data and expedited data constitute two distinct flows of data. If the normal flow is blocked, the user may nevertheless continue using the expedited one, but in sychronous mode a second process is needed.

Note that XTI supports two modes of sending data, record-oriented and stream-oriented. In the record-oriented mode, the concept of TSDU is supported, that is, message boundaries are preserved. In stream-oriented mode, message boundaries are not preserved and the concept of TSDU is not supported. A transport user can determine the mode by using the *t_getinfo*() function, and examining the *tsdu* field. If *tsdu* is greater than zero, this indicates that record-oriented mode is supported and the return value indicates the maximum TSDU size. If *tsdu* is zero, this indicates that stream-oriented transfer is supported. For more details see t_getinfo().

Overview of Connection Release

The ISO Connection-oriented Transport Service Definition supports only the abortive release. However, the TCP Transport Service Definition also supports an orderly release. So, some XTI implementations may support this orderly release.

An abortive release may be started from either the connection establishment phase or the data transfer phase. When in the connection establishment phase, a transport user may use the abortive release to reject a connect request. In the data transfer phase, either user may abort a connection at any time. The abortive release is not negotiated by the transport users and it takes effect immediately on request. The user on the other side of the connection is notified when a connection is aborted. The transport provider may also initiate an abortive release, in which case both users are informed that the connection no longer exists. There is no guarantee of delivery of user data once an abortive release has been initiated.

Whatever the state of transport connection, its user(s) will be informed as soon as possible of the failure of the connection through a disconnect event or an unsuccessful return from a blocking *t_snd*() or *t_rcv*() call. If the user wants to prevent loss of data by notifying the remote user of an imminent connection release, it is the user's responsibility to use an upper level mechanism. For example, the user may send specific (expedited) data and wait for the response of the remote user before issuing a disconnect request.

The orderly release capability is an optional feature of TCP. If supported by the TCP transport provider, orderly release may be started from the data transfer phase to enable two users to gracefully release a connection. The procedure for orderly release prevents the loss of data that may occur during an abortive release.

The functions that support connection release are:

t_snddis() This function can be issued by either transport user to initiate the abortive release of a transport connection. It may also be used to reject a connect request during the connection establishment phase.

t_rcvdis() This function identifies the reason for the abortive release of a connection, where the connection is released by the transport provider or another transport user.

t_sndrel() (OPTIONAL) This function can be called by either transport user to initiate an orderly release. The connection remains intact until both users call this function and *t_rcvrel*().

t_rcvrel() (OPTIONAL) This function is called when a user is notified of an orderly release request, as a means of informing the transport provider that the user is aware of the remote user's actions.

Overview of Connectionless Mode

The connectionless-mode transport service consists of two phases of communication: initialization/deinitialization and data transfer. A brief description of each phase and its associated functions is presented below. A state machine is described in the subsection Transport Layer Interface Sequence of Functions that defines the legal sequence in which functions from each phase may be issued.

In order to permit the transfer of connectionless data, a user (application) must:

1. Supply a transport endpoint for the appropriate type of provider (using *t_open*()); this establishes a transport endpoint in which the user may communicate with the provider.

2. Associate (bind) an address with this transport endpoint (using *t_bind*()).

3. The user may then send and/or receive connectionless data, as required, using the functions *t_sndudata*() and *t_rcvudata*(). Once the data transfer phase is finished, the application may either directly close the file descriptor returned by *t_open*() (using *t_close*()), thereby freeing the resource for future use, or start a new exchange of data after disassociating the old address and binding a new one.

Initialization/Deinitialization Phase

The functions that support the initialization/deinitialization tasks are the same functions used in the connection-mode service.

Overview of Data Transfer

Once a transport endpoint has been activated, a user is free to send and receive data units through that endpoint in connectionless mode as follows:

t_sndudata() This function enables transport users to send a self-contained data unit to the user at the specified protocol address.

t_rcvudata() This function enables transport users to receive data units from other users.

t_rcvuderr() This function enables transport users to retrieve error information associated with a previously sent data unit.

The only possible events reported to user are [T_UDERR], [T_DATA] and [T_GODATA]. Expedited data cannot be used with a connectionless transport provider.

Receiving Data

If data is available (a datagram or a part), the *t_rcvudata*() call returns immediately, indicating the number of octets received. If data is not immediately available, then the result of the *t_rcvudata*() call depends on the chosen mode:

- asynchronous mode:

  The call returns immediately, indicating failure. The user must either retry the call repeatedly, or "poll" for incoming data by using the *t_look*() function so as not to be blocked.

- synchronous mode:

  The call is blocked until one of the following conditions becomes true:

  - A datagram is received.
  - An error is detected by the transport provider.
  - A signal has arrived.

  The application may use the *t_look*() function to know if data is available instead of issuing a *t_rcvudata*() call which may be blocking.

Sending Data

- synchronous mode:

  In order to maintain some flow control, the *t_sndudata*() function returns when sending a new datagram becomes possible again. A process which sends data in synchronous mode may be blocked for some time.

- asynchronous mode:

  The transport provider may refuse to send a new datagram for flow control restrictions. In this case, the *t_sndudata*() call fails returning a negative value and setting *t_errno* to [TFLOW]. The user may retry later or use the *t_look*() function to be informed of the flow control restriction removal.

If *t_sndudata*() is called before the destination user has activated its transport endpoint, the data unit may be discarded.

Mandatory XTI Features

This section is concerned with the mandatory features of any implementation of XTI. Implementors of transport providers may use this section as a guide to let them know which requests must be implemented.

- The following functions, which correspond to the subset common to connection-oriented and connectionless services, are always implemented:

  t_bind()
  t_close()
  t_look()
  t_open()
  t_sync()
  t_unbind()

- If a Connection-oriented Transport Service is provided, then the following functions are always implemented:

  t_accept()
  t_connect()
  t_listen()
  t_rcv()
  t_rcvconnect()
  t_rcvdis()
  t_snd()
  t_snddis()

- If XTI supports the access to the Connectionless Transport Service, the following three functions are always implemented:

  t_rcvudata()
  t_rcvuderr()
  t_sndudata()

- Mandatory mechanisms:

  - Synchronous mode

  - Asynchronous mode

Transport Providers which provide only a Connectionless Transport Service may choose not to provide request codes for the functions which they need not support, and Transport Providers which provide only a Connection-oriented Transport Service may choose not to provide request codes for the functions which they need not support.

Optional XTI Features

This section lists those XTI features which are designated as being optional. An optional feature is defined as one whose implementation by a provider is not mandatory; consequently, the availability of such a feature cannot be guaranteed to applications.

- Optional functions:

t_error()	[Not implemented by CTOS at this time]
t_getinfo()	
t_getstate()	
t_optmgmt()	
t_alloc()	[Not implemented by CTOS at this time]
t_free()	[Not implemented by CTOS at this time]

NOTE: If a function is not implemented, then the name must be retained, but should return a value of -1 if started (except for *t_alloc*() which returns a null pointer), with *t_errno* set to [TNOTSUPPORT]).

- The orderly release mechanism (using *t_sndrel*() and *t_rcvrel*()) is optionally supported, although its use is makes applications not portable onto the ISO Transport Layer.

- Optional mechanisms:
  - the ability to manage (enqueue) more than one incoming connect indication at any one time.
  - automatic (default) generation of an address. This mechanism is not mandatory while no name server has been defined.

Transport Providers may choose not to provide request codes for the optional functions which they choose not to support.

XTI Functions Versus Protocols

The table below presents all the functions defined in XTI. Only the functions preceded by the character "M" are mandatory in XTI implementations. The optional functions are preceded by the character "O". Functions not implemented on CTOS are preceded by the character "U". The character "x" indicates that the mapping of that function is possible onto Connection-oriented or Connectionless Transport Service. The table indicates the type of utility functions as well.

		Necessary for Protocol		Utility Functions	
	Functions	**Connection Oriented**	**Connectionless**	**General**	**Memory**
M	*t_accept()*	x			
U	*t_alloc()*				x
M	*t_bind()*	x	x		
M	*t_close()*	x	x		
M	*t_connect()*	x			
U	*t_error()*			x	
U	*t_free()*				x
O	*t_getinfo()*			x	
O	*t_getstate()*			x	
M	*t_listen()*	x			
M	*t_look()*	x	x		
M	*t_open()*	x	x		
O	*t_optmgmt()*			x	
M	*t_rcv()*	x			
M	*t_rcvconnect()*	x			
M	*t_rcvdis()*	x			
O	*t_rcvrel()*	x			
M	*t_rcvudata()*		x		
M	*t_rcvuderr()*		x		
M	*t_snd()*	x			
M	*t_snddis()*	x			
O	*t_sndrel()*	x			
M	*t_sndudata()*		x		
M	*t_sync()*			x	
M	*t_unbind()*	x	x		

Classification of The XTI Functions

States and Events in XTI

The Figures 4-2 to 4-8 are included to describe the possible states of the transport provider as seen by the transport user, describe the incoming and outgoing events that may occur on any connection and identify the allowable sequence of function calls. Given a current state and event, the transition to the next state is shown as well as any actions that must be taken by the transport user.

The allowable sequence of functions is described in Figures 4-6, 4-7, and 4-8. The support functions, *t_getstate*(), *t_getinfo*(), *t_look*(), and *t_sync*() are excluded from the state tables because they do not affect the state of the interface. Each of these functions may be issued from any state except the uninitialized state.

Transport Interfaces States

XTI manages a transport endpoint by using at most eight states:

- T_UNINIT
- T_UNBND
- T_IDLE
- T_OUTCON
- T_INCON
- T_DATAXFER
- T_INREL
- T_OUTREL

The states T_OUTREL and T_INREL are significant only if the optional orderly release function is both supported and used.

Figure 4-2 describes all possible states of the transport provider as seen by the transport user. The service type may be connection mode, connection mode with orderly release, or connectionless mode.

State	Description	Service Type
T_UNINIT	uninitialized - initial and final state of interface	T_COTS T_CLTS T_COTS_ORD
T_UNBND	unbound	T_COTS T_COTS_ORD T_CLTS
T_IDLE	no connection established (this is the normal state for T_CLTS)	T_COTS T_COTS_ORD T_CLTS
T_OUTCON	outgoing connection pending for active user	T_COTS T_COTS_ORD
T_INCON	incoming connection pending for passive user	T_COTS T_COTS_ORD
T_DATAXFER	data transfer	T_COTS T_COTS_ORD
T_OUTREL	outgoing orderly release (waiting for orderly release indication)	T_COTS_ORD
T_INREL	incoming orderly release (waiting to send orderly release request)	T_COTS_ORD

Figure 4-2. Transport Interface States

Outgoing Events

The following outgoing events correspond to the successful return of the specified user-level transport functions, where these functions send a request or response to the transport provider. In Figure 4-3, some events (for instance, *acceptX*) are distinguished by the context in which they occur. The context is based on the values of the following:

ocnt count of outstanding connect indications (connect indications passed to the user but not accepted or rejected)

fd file descriptor of the current transport endpoint

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

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

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

Figure 4-3. Transport Interface Outgoing Events

Incoming Events

The following incoming events correspond to the successful return of the specified user-level transport functions, where these functions retrieve data or event information from the transport provider. One incoming event is not associated directly with the return of a function on a given transport endpoint:

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

In Figure 4-4, the *rcvdis* events are distinguished by the context in which they occur. The context is based on the value of *ocnt*, which is the count of outstanding connect indications on the current transport endpoint.

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

Figure 4-4. Transport Interface Incoming Events

Transport User Actions

Some state transitions are accompanied by a list of actions the transport user must take. These actions are represented by the notation [*n*], where *n* is the number of the specific action as described in Figure 4-5.

[1]	Set the count of outstanding connect indications to zero.
[2]	Increment the count of outstanding connect indications.
[3]	Decrement the count of outstanding connect indications.
[4]	Pass a connection to another transport endpoint as indicated in *t_accept*().

Figure 4-5. Transport Interface User Actions

State Tables

Figures 4-6, 4-7, and 4-8 describe the possible next states, given the current state and event. The state is that of the transport provider as seen by the transport user.

The contents of each box represent the next state given the current state (column) and the current incoming or outgoing event (row). An empty space represents a state/event combination that is invalid. Along with the next state, each space may include an action list (as specified in Figure 4-5). The transport user must take the specific actions in the order specified in the state table.

A separate table is shown for initialization/deinitialization, data transfer in connectionless mode and connection/release/data-transfer in connection mode.

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

Figure 4-6. Initialization/Deinitialization State Table

Event	T_IDLE State
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Figure 4-7. Data-Transfer State Table for Connectionless-Mode Service

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

Figure 4-8. Connection/Release/Data-Transfer State Table for Connection-Mode Service

Events and TLOOK Error Indication

The following list desribes the asynchronous events which cause an XTI call to return with a [TLOOK] error:

t_accept()	T_DISCONNECT, T_LISTEN
t_connect()	T_DISCONNECT, T_LISTEN
t_listen()	T_DISCONNECT
t_rcv()	T_DISCONNECT, T_ORDREL
t_rcvconnect()	T_DISCONNECT
t_rcvrel()	T_DISCONNECT
t_rcvudata()	T_UDERR
t_snd()	T_DISCONNECT, T_ORDREL
t_sndudata()	T_UDERR
t_unbind()	T_LISTEN
t_sndrel()	T_DISCONNECT

Once a [TLOOK] error has been received on a transport endpoint using an XTI function, subsequent calls to that and other XTI functions, to which the same [TLOOK] error applies, will continue to return [TLOOK] until the event is consumed. An event causing the [TLOOK] error can be determined by calling *t_look*() and then can be consumed by calling the corresponding consuming XTI function as defined in the table in the section Events and tlook(), earlier in this chapter.

Transport Protocol-Specific Options

In order to maintain protocol independence the transport specific options and structures are not generally defined. They are, however, defined for each particular transport provider (for instance, TCP or ISO) within the appropriate appendix.

The interface allows the transport user to set protocol-specific options through the *t_connect*(), *t_accept*(), and *t_listen*() function calls. Each of these functions provide an *opt* parameter which is a **struct netbuf**. The **netbuf** structure contains:

Offset	Field	Size (Bytes)	Contents
0	maxlen	2	max buffer length
2	len	2	length of data in buffer
4	pBuf	4	pointer to data buffer

To pass options through this interface, the *pBuf* field of the **netbuf** structure must point to the option structure specified for the transport provider (see the appropriate appendix).

Connection Mode Options

In connection mode, an option structure is defined which contains those parameters needed to establish a transport connection. Connection parameters comprise two main categories: service related parameters and management related parameters (or protocol option parameters).

The service related parameters are those parameters which define the quality of the transport service that the user wishes to obtain. These parameters are not mandatory and the user may omit them if there is no particular requirement. If the transport provider cannot offer the required quality of service, it will refuse either to establish or accept the connection. If the requirements of the user concerning the quality of service are too high, the establishment of the connection may be refused either by the local transport provider, the remote transport provider, or the remote end user.

The management parameters define some management features needed by the transport provider to establish the transport connection. These parameters are strongly bound to the local environment of the application. These parameters depend on the location of the two transport endpoints and the type of network being used. In most applications, management parameters should be hidden from the user.

Normally, the transport user should only specify the service related parameters and not the management parameters. It is meaningless to specify both categories of parameters at the same time.

Connectionless Mode Options

In connectionless mode an option structure, which contains parameters describing the required quality of service is defined. These parameters are optional and the transport user may omit to define them if there is no particular requirement. Note that if the transport provider cannot offer the quality of service which is indicated as mandatory by the user, it will refuse to send the associated data.

CTOS Support of XTI Procedures

Support of Optional Procedures

The X/Open Transport Interface specification describes, in addition to the procedures given below, the optional procedures *t_alloc*(), *t_free*(), and *t_error*(). These procedures are also regarded as optional in the CTOS/Open Transport Layer API. An X/Open compliant interface is required to retain these procedures by name, but is permitted to return a value of -1 if used, with *t_errno* set to [TNOTSUPPORT]. The exception to this is *t_alloc*(), which returns a null pointer if unimplemented.

For CTOS product implementors who wish to provide these procedures with the normal CTOS calling convention rather than the C calling convention, the following information is provided:

t_alloc() takes three word parameters and returns a pointer.

t_free() takes one pointer parameter and one word parameter, and returns a word.

t_error() takes one pointer parameter and returns a word.

Support of Multiple Procedures by a Single Request

XTI library functions do not map one-for-one onto their underlying requests. In some cases, multiple library functions make use of the same request. The cases where this occurs are summarized here:

t_connect() and *t_rcvconnect*() are implemented as a single request which is issued by the Transport Client when *t_connect*() is called. If this is in asynchronous mode, the XTI library procedure returns before the request is responded to by the Transport Provider, which responds only when a the connection attempt has been successful or an error has occurred. After the response comes back, the Transport Client can call *t_rcvconnect*() to determine whether the connection attempt has been successful.

t_sync(), *t_getinfo*(), and *t_optmgmt*() are implemented as a single request which returns both sets of data. *t_sync*() and *t_getinfo*() share the same request block structure as well as the same request code. *t_optmgmt*() uses the same request code but has a different request block structure.

t_look() and *t_getstate*() are implemented as a single request which returns both sets of data.

t_rcvdis() and *t_rcvrel*() are implemented as a single request with an additional parameter to tell them apart.

t_snddis() and *t_sndrel*() are implemented as a single request with an additional parameter to tell them apart.

t_sndudata() and *t_rcvuderr*() are implemented as a single request which is issued by the Transport Client when *t_sndudata*() is called. The XTI library procedure returns before the request is responded to by the Transport Provider, which responds only after an error has occurred, or no more errors will be reported to it by the network. After the response comes back, the Transport Client can call *t_rcvuderr*() to find out what error occurred.

Use of Multiple Requests by a Single Procedure

In addition, the *t_open*() function issues two requests: the request associated directly with *t_open*(), and the *t_getinfo*()/*t_sync*() request.

Parameter Definition File for the Transport Layer

A Transport Client calls *t_open* to establish connection with the Transport Provider. Transport dependencies are removed from the parameters of these procedures by placing them in a Parameter Definition File. The parameters in the file are dependent upon the type of the Transport Provider. Every parameter in the file is terminated by a new line character (hexadecimal 0A). Parameter labels, or comments, delimited by a pair of colons are allowed in the file but not in the string if passed directly to the Transport Provider (the Protocol Manager automatically deletes comments).

The only required information is the Transport Provider Name (excluding the Device Specification which is supplied by the Protocol Manager) which is the first entry in the file or string.

The following example is a Parameter Definition File for an ISO Transport Layer, as used by an ISO Session Layer client:

```
:Transport Layer Name:ISO TP4
```

More parameters could be added to the PDF to accomodate implementation-dependent requirements.

XTI Library Functions and Parameters

This section first discusses several conventions used to describe the library routines, and then discusses the library routines and their underlying requests.

Key for Parameter Arrays

For each XTI library function description, 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.
x(x)	The parameter (a pointer) is meaningful because it points to a meaningful object.
?(?)	The parameter (a pointer) is meaningful if non-zero, and the object to which it points is meaningful if present.

Return of TLOOK Error

Many of the XTI functions contained in this chapter return a [TLOOK] error to report the occurrence of an asynchronous event. For these functions a complete list describing the function and the events is provided in the subsection entitled Events and TLOOK Error Indication, earlier in this section.

Description

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

t_accept 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 *pCall* points to a structure which contains information required by the transport provider to complete the connection.

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

A transport user may accept a connection on either the same or 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* equals *fd*), the user must have responded to any previous connect indications received on that transport endpoint (using *t_accept* or *t_snddis*). Otherwise, *t_accept* will fail and set *t_errno* to [TBADF].

If a different transport endpoint is specified (*resfd* not equal to *fd*), the endpoint must be bound to a protocol address (if it is the same, *qlen* must be set to 0) and must be in the T_IDLE state (see *t_getstate*) before the *t_accept* is issued.

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

The values of parameters specified by *opt* and the syntax of those values are protocol-specific. The *udata* argument enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider as returned 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.

Procedural Interface

t_accept (fd, resfd, pCall): Integer

where

fd

is the file descriptor returned by *t_open* for the original local transport endpoint. This should be the file descriptor used in the *t_listen* procedure which returned with notification of the call now being accepted.

resfd

is the file descriptor where the connection is to be established. This should be another file descriptor returned by *t_open*. It can be the same as *fd*.

pCall

is a pointer to a structure *call* of type **t_call** which contains the following fields:

Offset	Field	Length
0	addr.maxlen	2
2	addr.len	2
4	addr.pBuf	4
8	opt.maxlen	2
10	opt.len	2
12	opt.pBuf	4
16	udata.maxlen	2
18	udata.len	2
20	udata.pBuf	4
24	sequence	2

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	10
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	1
16	reserved	2	
18	resfd	2	
20	call.sequence	2	
22	call.addr.pBuf	4	
26	call.addr.len	2	
28	call.opt.pBuf	4	
32	call.opt.len	2	
34	call.udata.pBuf	4	
38	call.udata.len	2	

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

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

[TOUTSTATE] The function was 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.

Return Value

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_connect, t_getstate, t_listen, t_open, t_optmgmt, t_rcvconnect.

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	x	(x)
ret.qlen	/	x>=0

The *t_bind* 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 queuing 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 *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 *addr*, be bound to the given transport endpoint. The parameter *len* specifies the number of bytes in the address and *pBuf* 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 may be different from the address specified by the user in *req*). In *ret*, the user specifies *maxlen*, which is the maximum size of the address buffer, and *pBuf*, 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 *pBuf* points to the bound address. If *maxlen* is not large enough to hold the returned address, an error will result.

If the requested address is not available, or if no address is specified in *req* (the *len* field of addr in *req* is zero), the transport provider will assign an appropriate address to be bound only if automatic generation of an address is supported, and will return that address in the *addr* field of *ret*. The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested. If in any XTI implementation the *t_bind* function does not allocate a local transport address, then the returned address is always the same as the input address and the structure *req.addr* must be filled by the user before the call, otherwise, if the local address is not provided for the call (*req.addr.len* equals *0*), *t_bind* will return -1 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 must 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 initializing a connection-mode service. It specifies the number of outstanding connect indications and 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. On return, the *qlen* field in *ret* will contain the negotiated value.

This function allows more than one transport endpoint to be bound to the same protocol address (however, the transport provider must 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, the transport provider will assign another address to be bound to that endpoint or, if automatic generation of addresses is not supported, 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.

Procedural Interface

t_bind (fd, pReq, pRet): Integer

where

fd

is the file descriptor returned by *t_open*.

pReq,
pRet

are pointers to structures *req* and *ret*, of type **t_bind**, which contain the following fields:

Offset	Field	Length
0	addr.maxlen	2
2	addr.len	2
4	addr.pBuf	4
8	qlen	2

where

pret.addr.len
sret.addr.len

describe a word where the length of the address chosen by the transport provider is returned.

pret.qlen
sret.qlen

describe a word where the queue length chosen by the transport provider is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	2
16	reserved	2	
18	req.qlen	2	
20	req.addr.pBuf	4	
24	req.addr.len	2	
26	ret.addr.pBuf	4	
30	ret.addr.maxlen	2	
32	pret.addr.len	4	
36	sret.addr.len	2	2
38	pret.qlen	4	
42	sret.qlen	2	2

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

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

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, the procedural interface returns a value of -1 and sets *t_errno* to indicate an error.

See Also

t_close, t_open, t_optmgmt, t_unbind.

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 descriptor associated with the transport endpoint.

The function *t_close* should be called from the T_UNBIND 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, the file descriptor will be closed. 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.

Procedural Interface

t_close (fd): Integer

where

fd

is the file descriptor returned by *t_open*.

Request Block

This request closes the file descriptor and all routes which it may hold open.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

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

Return Value

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.

Description

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

The **netbuf** structure *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, and *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 described for ISO and TCP protocols in Appendixes C, D and E. 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 the corresponding options structures (**isoco_options, isocl_options** or **tcp_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 (for instance, a return value of zero) indicates that the requested connection has been established. However, if O_NONBLOCK is set (using *t_open*), *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.

Procedural Interface

t_connect (fd, pSndcall, pRcvcall): Integer

where

fd

is the file descriptor returned by *t_open*.

pSndcall
pRcvcall

are pointers to structures *sndcall* and *rcvcall*, of type **t_call**, which contain the following fields:

Offset	Field	Length
0	addr.maxlen	2
2	addr.len	2
4	addr.pBuf	4
8	opt.maxlen	2
10	opt.len	2
12	opt.pBuf	4
16	udata.maxlen	2
18	udata.len	2
20	udata.pBuf	4
24	sequence	2

Request Block

The request for this procedure is the same as for *t_rcvconnect*. *t_connect* will, in asynchronous mode, return before the Transport Provider responds to this request.

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	3
16	reserved	2	
18	sndcall.addr.pBuf	4	
22	sndcall.addr.len	2	
24	sndcall.opt.pBuf	4	
28	sndcall.opt.len	2	
30	sndcall.udata.pBuf	4	
34	sndcall.udata.len	2	
36	rcvcall.addr.pBuf	4	
40	rcvcall.addr.maxlen	2	
42	rcvcall.opt.pBuf	4	
46	rcvcall.opt.maxlen	2	
48	rcvcall.udata.pBuf	4	
52	rcvcall.udata.maxlen	2	

The pb/cb pairs

rcvcall.addr.pBuf/rcvcall.addr.maxlen
rcvcall.opt.pBuf/rcvcall.opt.maxlen
rcvcall.udata.pBuf/rcvcall.udata.maxlen

all describe a structure as follows:

Offset	Field	(Bytes)
0	cbRet	2
2	data	X

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

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

Return Value

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_accept, t_getinfo, t_listen, t_open, t_optmgmt, t_rcvconnect.

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

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

The values of the fields in *info* have the following meanings:

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

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

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

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

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

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

servtype This field specifies the service type supported by the transport provider.

If a transport user is concerned with protocol independence, these fields may be accessed to determine how large the buffers must be to hold each piece of information. 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. The function *t_getinfo* enables a user to retrieve the current characteristics of the underlying transport protocol.

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*.

Procedural Interface

t_getinfo (fd, pInfoRet): Integer

where

fd

is the file descriptor returned by *t_open*.

pInfoRet

is a pointer to a structure *info* of type **t_info** which contains the following fields:

Offset	Field	Size (Bytes)	Contents
0	addr	4	max size of the transport protocol address
4	options	4	max number of bytes of protocol-specific options
8	tsdu	4	max size of a transport service data unit (TSDU)
12	etsdu	4	max size of an expedited transport service data unit (ETSDU)
16	connect	4	max amount of data allowed on connection establishment functions
20	discon	4	max amount of data allowed on t_snddis and t_rcvdis functions
24	servtype	4	service type supported by the transport provider

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	1	13
16	reserved	2	
18	plnfoRet	4	
22	slnfoRet	2	

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TSYSERR]	A system error has occurred during execution of this function.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.

Return Value

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_open.

Description

Parameters	Before call	After call
fd	x	/

Procedural Interface

t_getstate (fd): Integer

where

fd

is the file descriptor returned by *t_open*.

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

Request Block

NOTE: This is the same request (and same opcode) as *t_look*.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	4
16	reserved	2	
18	pRetArea	4	
22	sRetArea	2	4

The pb/cb pair pRetArea/sRetArea describes a structure as follows:

Offset	Field	Size (Bytes)
0	LookEventCode	2
2	StateCode	2

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[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 provider is undergoing a transient state change.

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

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

Return Value

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.

Description

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

The *t_listen* 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.

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

Since this function returns values for the *addr*, *opt* and *udata* fields of *call*, the *maxlen* 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 using *t_open*, *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].

Procedural Interface

t_listen (fd, pCall): Integer

where

fd

is the file descriptor returned by *t_open*.

pCall

is a pointer to a structure *call* of type **t_call** which contains the following fields:

Offset	Field	Length
0	addr.maxlen	2
2	addr.len	2
4	addr.pBuf	4
8	opt.maxlen	2
10	opt.len	2
12	opt.pBuf	4
16	udata.maxlen	2
18	udata.len	2
20	udata.pBuf	4
24	sequence	2

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	4
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	5
16	reserved	2	
18	call.addr.pBuf	4	
22	call.addr.maxlen	2	
24	call.opt.pBuf	4	
28	call.opt.maxlen	2	
30	call.udata.pBuf	4	
34	call.udata.maxlen	2	
36	pcall.sequence	4	
40	scall.sequence	2	2

where

pcall.sequence
scall.sequence

describe a word where the sequence number chosen by the transport provider is returned.

The pb/cb pairs

call.addr.pBuf/call.addr.maxlen
call.opt.pBuf/call.opt.maxlen
call.udata.pBuf/call.udata.maxlen

all describe a structure as follows:

Offset	Field	Size (Bytes)
0	cbRet	2
2	data	X

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[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 is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect indication information to be returned in *call* is discarded. The value of *sequence* returned can be used to do a *t_snddis*.

[TNODATA] O_NONBLOCK was set, but no connect indications have 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.

Return Value

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_accept, t_bind, t_connect, t_open, t_optmgmt, t_rcvconnect.

Description

Parameters	Before call	After call
fd	x	/

The *t_look* 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 the section Events and TLOOK Error Indication.

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

Procedural Interface

t_look (fd): Integer

where

fd

is the file descriptor returned by *t_open*.

Request Block

NOTE: This is the same request (and same opcode) as *t_getstate*.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	4
16	reserved	2	
18	pRetArea	4	
22	sRetArea	2	4

The pb/cb pair *pRetArea/sRetArea* describes a structure as follows:

Offset	Field	Size (Bytes)
0	LookEventCode	2
2	StateCode	2

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

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

Return Value

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 have been lifted. Normal data may be sent again.
T_GOEXDATA	Flow control restrictions on expedited data flow have been lifted. Expedited data may be sent again.

On failure, -1 is returned and *t_errno* is set to indicate an error. If the request is being used directly, rather than the XTI library interface, the following event may also be returned:

T_SENDSYNC	This is a request by the Transport Service Provider to the transport client to send sync data so that another user of this transport endpoint can be synchronized.

See Also

t_open, t_snd, t_sndudata.

Description

Parameters	Before call	After call
pbProviderName	x	/
oflag	x	/
info.addr	/	x
info.options	/	x
info.tsdu	/	x
info.etsdu	/	x
info.connect	/	x
info.discon	/	x
info.servtype	/	x

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

The argument *pbProviderName* points to a transport provider identifier, which is of one of three types. The default type, if neither O_FILESPEC nor O_NAME_PDS is set in *oflag*, is the name by which the Transport Service Provider is known to the Protocol Manager. If O_FILESPEC is set, the argument *pbProviderName* points to a file specification for a Parameter Definition File. This file, examples of which are given in the section Parameter Definition File for the Transport Layer, contains a Transport Provider Name and optional parameter data. If O_NAME_PDS is set, the argument *pbProviderName* points to a string of the following format:

Offset	Field	Size (Bytes)	Contents
0	cbName	1	length of the transport provider name
1	Name	X	transport provider name
1+X	cbParams	2	length of parameter definition string
3+X	cbParam1	1	length of first parameter
4+X	Param1	Y	first parameter
4+X+Y	cbParam2	1	length of second parameter
5+X+Y	Param2	Z	second parameter

where additional parameters can be added as needed to the end of the structure.

The argument *oflag* identifies any open flags. *oflag* is constructed from O_RDWR optionally bitwise inclusive-or'ed with O_NONBLOCK, O_FILESPEC, and O_NAME_PDS. *oflag* has significance only to the XTI library; it is not passed on to the Transport Provider.

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. The values of the fields have the following meanings:

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

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

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

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

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

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

servtype This field specifies the service type supported by the transport provider.

If a transport user is concerned with protocol independence, these fields may be accessed to determine how large the buffers must be to hold each piece of information. 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 these three services at one time.

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

Procedural Interface

t_open (pbProviderName, oflag, pbrgInfo): Integer

where

pbProviderName

is a pointer to a null-terminated string containing either a Transport Service Provider Name; a file specification for a transport parameter definition file; or a Transport Service Provider Name plus a parameter definition string, depending on the value of *oflag*.

oflag

is a word.

pbrgInfo

is a pointer to an array *info* of type **t_info**, which contains the following fields:

Offset	Field	Size (Bytes)	Contents
0	addr	4	max size of the transport protocol address
4	options	4	max number of bytes of protocol-specific options
8	tsdu	4	max size of a transport service data unit (TSDU)
12	etsdu	4	max size of an expedited transport service data unit (ETSDU)
16	connect	4	max amount of data allowed on connection establishment functions
20	discon	4	max amount of data allowed on *t_snddis* and *t_rcvdis* functions
24	servtype	4	service type supported by the transport provider

Request Block

The request block for *t_open* does not obtain the **t_info** data. Instead, the XTI library issues both the *t_open* request and the *t_getinfo* request when a call to the *t_open* function is made.

Although this request is routed by Device Spec, no Device Spec is supplied to *t_open*. Prior to issueing the *t_open* request, the XTI library issues a *RequestServiceProvider* request to the Protocol Manager. The Device Spec for the Transport Service Provider is returned by the Protocol Manager.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	Reserved	6	
18	pbProviderDeviceSpec	4	
22	cbProviderDeviceSpec	2	
24	pbClientName	4	
28	cbClientName	2	
30	pPDString	4	
34	cPDString	2	
36	pbFdRet	4	
40	cbFdRet	2	2

pbDeviceSpec/cbDeviceSpec

describe a device specification for the desired Transport Service Provider. If not supplied by the Transport Client, *cbProviderDeviceSpec* should be set to zero.

pbClientName/cbClientName

describe a string where the Transport Client can place its own name. This string can be up to twelve bytes long. This string is optional. If not supplied by the Transport Client, *cbClientName* should be set to zero.

pPDString/cPDString

describe a string of Parameter Definition File data, in the format returned by RequestTransportProvider. (This string may not be required by some Transport Providers.)

pbFdRet/cbFdRet

describe a word where the file descriptor of the connection being opened is returned. This is the function return value.

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[TBADFLAG]	An invalid flag is specified.
[TBADNAME]	Invalid transport provider name.
[TSYSERR]	A system error has occurred during execution of this function.

Return Value

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.

Description

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

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

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	x	(x)
ret.flags	/	x

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, *pBuf* 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 *pBuf* 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. The actual structure and content of the options is imposed by the transport provider.

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

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

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

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

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

Procedural Interface

t_optmgmt (fd, pReq, pRet): Integer

where

fd

is the file descriptor returned by *t_open*.

pReq
pRet

are pointers to structures *req* and *ret*, of type **t_optmgmt**, which contain the following fields:

Offset	Field	Length
0	opt.maxlen	2
2	opt.len	2
4	opt.pBuf	4
8	flags	4

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	7
16	reserved	2	
18	req.flags	2	
20	req.opt.pBuf	4	
24	req.opt.len	2	
26	ret.opt.pBuf	4	
30	ret.opt.maxlen	2	
32	ret.opt.plen	4	
36	ret.opt.slen	2	2
38	pFlagsRet	4	
42	sFlagsRet	2	2

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[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 is specified.
[TBUFOVFLW]	The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.
[TSYSERR]	A system error has occurred during execution of this function.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.

Return Value

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_accept, t_connect, t_getinfo, t_listen, t_open, t_rcvconnect.

Description

Parameters	Before call	After call
fd	x	/
pBufRet	x	(x)
sBufMax	x	/
pFlagsRet	/	x

The *t_rcv* function receives either normal or expedited data. The argument *fd* identifies the local transport endpoint through which data will arrive, *pBufRet* points to a receive buffer where user data will be placed and *sBufMax* specifies the size of the receive buffer. The argument *pflagsRet* points to *flags*, which may be set on return from *t_rcv* and specifies optional flags.

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 (using *t_open*), *t_rcv* will execute in asynchronous mode and will fail if no data is available. (See [TNODATA].)

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

On return, the data returned is expedited data if T_EXPEDITED is set in *flags*. If the number of bytes of expedited data exceeds *sBufMax*, *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.

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.

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.

Procedural Interface

t_rcv (fd, pBufRet, sBufRet, pFlagsRet): Integer

where

fd

is the file descriptor returned by *t_open*.

pBufRet
sBufMax

describe the buffer into which receive data is to be returned.

pFlagsRet

is a pointer to a word where a series of bit flags, OR-ed together, is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	16
16	reserved	2	
18	pBufRet	4	
22	sBufMax	2	
24	pFlagsRet	4	
28	sFlagsRet	2	2

The pb/cb pair *pBufRet/sBufMax* describes a structure as follows:

Offset	Field	Size (Bytes)
0	cbRet	2
2	data	X

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[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.

Return Value

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

See Also

t_getinfo, t_look, t_open, t_snd.

Description

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

The *t_rcvconnect* 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.

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

The *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *pCall* 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 (using *t_open*), *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].) In this case, *t_rcvconnect* must be called again to complete the connection establishment phase and retrieve the information returned in *call*.

Procedural Interface

t_rcvconnect (fd, pCall): Integer

where

fd

is the file descriptor returned by *t_open*.

pCall

is a pointer to a structure of type **t_call**, which contains the following fields:

Offset	Field	Length
0	addr.maxlen	2
2	addr.len	2
4	addr.pBuf	4
8	opt.maxlen	2
10	opt.len	2
12	opt.pBuf	4
16	udata.maxlen	2
18	udata.len	2
20	udata.pBuf	4
24	sequence	2

Request Block

The request block for this procedure is the same as for *t_connect*, which returns before the response to its request. This procedure returns the response to the request.

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

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

[TBUFOVFLW] The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument and the connect information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to T_DATAXFER.

[TNODATA] O_NONBLOCK was set, but a connect confirmation has not yet arrived.

[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.

Return Value

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_accept, t_bind, t_connect, t_listen, t_open, t_optmgmt.

Description

The *t_rcvdis* 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.

Parameters	Before call	After call
fd	x	/
discon.udata.maxlen	x	/
discon.udata.len	/	x
discon.udata.pBuf	x	(?)
discon.reason	/	x
discon.sequence	/	?

The field *reason* specifies the field 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 now 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 (using *t_listen*) and *discon* is a null pointer, the user will be unable to identify with which connect indication the disconnect is associated.

Procedural Interface

t_rcvdis (fd, pDiscon): Integer

where

fd

is the file descriptor returned by *t_open*.

pDiscon

is a pointer to a structure *discon* of type **t_discon**, which contains the following fields:

Offset	Field	Length
0	udata.maxien	2
2	udata.len	2
4	udata.pBuf	4
8	reason	2
10	sequence	2

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	8
16	reserved	2	
18	discon.udata.pBuf	4	
22	discon.udata.maxlen	2	
24	pdiscon.udata.len	4	
28	sdiscon.udata.len	2	2
30	pbStructRet	4	
34	cbStructRet	2	4

where

pdiscon.udata.len
sdiscon.udata.len

describe a word where the count of bytes actually received is returned by the transport provider.

The pb/cb pair pbStructRet/cbStructRet describe a structure as follows:

Offset	Field	Size (Bytes)
0	reason	2
2	sequence	2

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

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

[TNODIS] No disconnect indication currently exists on the specified transport endpoint.

[TBUFOVFLW] The number of bytes allocated for incoming data is not sufficient to store the data. 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*.

Return Value

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_connect, t_listen, t_open, t_snddis.

Description

Parameters	Before call	After call
fd	x	/

The *t_rcvrel* 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*.

Procedural Interface

t_rcvrel (fd): Integer

where

fd

is the file descriptor returned by *t_open*.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	9
16	reserved	2	
18	discon.udata.buf	4	
22	discon.udata.maxlen	2	
24	pdiscon.udata.len	4	
28	sdiscon.udata.len	2	2
30	pbStructRet	4	
34	cbStructRet	2	4

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

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

Return Value

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_sndrel.

Description

Parameters	Before call	After call
fd	x	/
unitdata.addr.maxlen	x	/
unitdata.addr.len	/	x
unitdata.addr.buf	x	(x)
unitdata.opt.maxlen	x	/
unitdata.opt.len	/	x
unitdata.opt.buf	x	(x)
unitdata.udata.maxlen	x	/
unitdata.udata.len	/	x
unitdata.udata.buf	x	(x)
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 *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 protocol-specific options that were associated with this data unit, and *udata* specifies the user data that was received.

By default, *t_rcvudata* operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if O_NONBLOCK is set (using *t_open*), *udata* 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.

Procedural Interface

t_rcvudata (fd, pUnitdata, pflags): Integer

where

fd

is the file descriptor returned by *t_open*.

pUnitdata

is a pointer to a structure *unitdata* of type **t_unitdata**, which contains the following fields:

Offset	Field	Length
0	udata.maxien	2
0	addr.maxlen	2
2	addr.len	2
4	addr.pBuf	4
8	opt.maxlen	2
10	opt.len	2
12	opt.pBuf	4
16	udata.maxlen	2
18	udata.len	2
20	udata.pBuf	4

pflags

is a pointer to a word where a series of bit flags, OR-ed together, is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	4
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	17
16	reserved	2	
18	unitdata.addr.pBuf	4	
22	unitdata.addr.maxlen	2	
24	unitdata.opt.pBuf	4	
28	unitdata.opt.maxlen	2	
30	unitdata.udata.pBuf	4	
34	unitdata.udata.maxlen	2	
36	pflags	4	
40	sflags	2	2

The pb/cb pairs

unitdata.addr.buf/unitdata.addr.maxlen
unitdata.opt.buf/unitdata.opt.maxlen
unitdata.udata.buf/unitdata.udata.maxlen

all describe a structure as follows:

Offset	Field	Size (Bytes)
0	cbRet	2
2	data	X

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[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 is not sufficient to store the information. The unit data information to be returned in *unitdata* will be discarded.

[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[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.

Return Value

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_open, t_rcvuderr, t_sndudata.

Description

Parameters	Before call	After call
fd	x	/
uderr.addr.maxlen	x	/
uderr.addr.len	/	x
uderr.addr.buf	x	(x)
uderr.opt.maxlen	x	/
uderr.opt.len	/	x
uderr.opt.buf	x	(x)
uderr.error	/	x

The *t_rcvuderr* 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. 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 protocol-specific options that were associated with the data unit and *error* specifies a protocol dependent error code.

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

Procedural Interface

t_rcvuderr (fd, pUderr): Integer

where

fd

is the file descriptor returned by *t_open*.

pUderr

is a pointer to a structure *uderr* of type **t_uderr**, which contains the following fields:

Offset	Field	Length
0	addr.maxlen	2
2	addr.len	2
4	addr.pBuf	4
8	opt.maxlen	2
10	opt.len	2
12	opt.pBuf	4
16	error	4

Request Block

The request block for this procedure is the same as for *t_sndudata*, which returns before the response to its request. This procedure returns the response to the request.

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TNOUDERR]	No unit data error indication currently exists on the specified transport endpoint.
[TBUFOVFLW]	The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data error information to be returned in *uderr* will be discarded.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

Return Value

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_rcvudata, t_sndudata.

Description

The *t_snd* function is used to send either normal or expedited data.

Parameters	Before call	After call
fd	x	/
pBuf	x(x)	/
sBuf	x	/
flags	x	/

The argument *fd* identifies the local transport endpoint over which data should be sent, *pBuf* points to the user data, *sBuf* specifies the number of bytes of user data to be sent and *flags* specifies any optional flags:

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. 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_geninfo*, the T_MORE flag is not meaningful and should be ignored.

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 (using *t_open*), *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 using *t_look*.

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 *sBuf*.

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 *sBuf*. If *sBuf* 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 of *t_open* or *t_getinfo*. Failure to comply will result in protocol error. (See [TSYSERR].)

The error [TLOOK] may be returned to inform the process that an event (for instance, a disconnect) has occurred.

Procedural Interface

t_snd (fd, pBuf, sBuf, flags): Integer

where

fd

is the file descriptor returned by *t_open*.

pBuf
sBuf

describe the data to be transmitted.

flags

is a word which is the OR-ed together value of a set of bit flags.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	reserved	3	
17	flags	1	
18	pBuf	4	
22	sBuf	2	
24	pcbRet	4	
28	scbRet	2	2

pcbRet/scbRet

describe a word where the count of bytes actually transmitted is returned. This is the function return value for *t_snd*.

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TBADDATA]	Illegal amount of data; zero octets is not supported.
[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. A protocol error may not cause *t_snd* to fail until a subsequent access of the transport endpoint.

Return Value

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.

NOTE: 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.

See Also

t_getinfo, t_open, t_rcv.

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	?	/

The *t_snddis* function is used to initiate an abortive release on an already established connection or to reject a connect request. The argument *fd* identified the local transport endpoint of the connection, and *call* specifies information associated with the abortive release.

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.

Procedural Interface

t_snddis (fd, pCall) : Integer

where

fd

is the file descriptor returned by *t_open*.

pCall

is a pointer to a structure *call* of type **t_call**, which contains the following fields:

Offset	Field	Length
0	addr.maxlen	2
2	addr.len	2
4	addr.pBuf	4
8	opt.maxlen	2
10	opt.len	2
12	opt.pBuf	4
16	udata.maxlen	2
18	udata.len	2
20	udata.pBuf	4
24	sequence	2

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	10
16	reserved	2	
18	call.sequence	2	
20	call.udata.buf	4	
24	call.udata.len	2	

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[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. Some outbound data queued for this endpoint may be lost.
[TBADSEQ]	An invalid sequence number was specified, or a null *call* pointer was specified when rejecting a connect request. Some outbound data queued for this endpoint may be lost.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TSYSERR]	A system error has occurred during execution of this function.

Return Value

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_connect, t_getinfo, t_listen, t_open.

Description

Parameters	Before call	After call
fd	x	/

The *t_sndrel* 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*.

Procedural Interface

t_sndrel (fd) : Integer

where

fd

is the file descriptor returned by t_open.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	11
16	reserved	2	

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[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.

Return Value

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.

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)	/

The *t_sndudata* 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.

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

If the *len* field of *udata* is zero, and sending of zero octets is not supported by the underlying transport service, *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 (using *t_open*), *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 using *t_look*.

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_geninfo*, the provider will generate a protocol error. (See [TSYSERR].) If *t_sndudata* is called before the destination user has activated its transport endpoint (see *t_bind*), the data unit may be discarded.

The request block for *t_sndudata* may be outstanding at the transport provider for an indeterminate amount of time. The transport provider keeps the request until all possible errors are available to it.

Procedural Interface

t_sndudata (fd, pUnitdata) : Integer

where

fd

is the file descriptor returned by *t_open*.

pUnitdata

is a pointer to a structure *unitdata* of type **t_unitdata**, containing the following fields:

Offset	Field	Length
0	addr.maxlen	2
2	addr.len	2
4	addr.pBuf	4
8	opt.maxlen	2
10	opt.len	2
12	opt.pBuf	4
16	udata.maxlen	2
8	udata.len	2
20	udata.pBuf	4

Request Block

This is the same request code as is used tor *t_rcv* and *t_rcvudata*.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	18
16	reserved	2	
18	unitdata.addr.pBuf	4	
22	unitdata.addr.len	2	
24	unitdata.opt.pBuf	4	
28	unitdata.opt.len	2	
30	unitdata.udata.pBuf	4	
34	unitdata.udata.len	2	
36	puderr.error	4	
40	suderr.error	2	4

where

puderr.error
suderr.error

describe a double word area where the *error* field for *t_rcvuderr* is returned.

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[TBADDATA]	Illegal amount of data; zero octets is not supported.
[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. A protocol error may not cause *t_sndudata* to fail until a subsequent access of the transport endpoint.

Return Value

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_open, t_rcvudata, t_rcvuderr.

Description

Parameters	Before call	After call
fd	x	/

For the transport endpoint specified by *fd*, *t_sync* synchronizes the data structures managed by the transport library with information from another user of the same transport endpoint. In doing so, it can convert an uninitialized file descriptor (possessed by another CTOS task or possibly obtained using POSIX *fork* and *exec* functions) to an initialized transport endpoint, assuming that file descriptor referenced a transport endpoint, by updating the necessary library data structures. This function allows two cooperating processes to synchronize their interaction with a transport provider.

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

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the 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.

Procedural Interface

t_sync (fd) : Integer

where

fd

is the file descriptor returned by *t_open*.

Request Block

This is the same request block and request code used for *t_getinfo*. It also shares the same request code as the *t_optmgmt* request. *t_sync* uses two formats for this request.

Format One: Retrieval of sync data. This request causes the Transport Provider to issue a T_SENDSYNC event to the oldest other user of this transport endpoint.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	14
16	reserved	2	
18	psyncdataret	4	
22	ssyncdataret	2	

where

opcode

is a word. The value of 14 indicates that this request is for the *t_sync* retrieval function.

pSyncDataRet
sSyncDataRet

describe a buffer where the complete state of this transport endpoint, as seen by another user of this endpoint, is returned.

Format Two: Sending of sync data. The XTI library issues this request in response to receipt of a T_SENDSYNC event (which is never passed on to the library user).

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	15
16	reserved	2	
18	psyncdata	4	
22	ssyncdata	2	

where

opcode

is a word. The value of 15 indicates that this request is for the *t_sync* data send function.

pSyncData
sSyncData

describe a buffer where this user sends the state of this transport endpoint, as it sees it, to the transport service provider. The format of this data is transparent to the transport service provider.

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[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.

Return Value

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_UNBIND 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)

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.

Procedural Interface

t_unbind (fd) : Integer

where

fd

is the file descriptor returned by *t_open*.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	
12	fd	2	
14	opcode	2	12
16	reserved	2	

Errors

On failure, the procedural interface sets *t_errno* equal to one of the following errors (which are returned in the *ercRet* field of the request block):

[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.

Return Value

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.

Using the Transport Layer Interface

Transport Layer Interface Sequence of Functions

In order to describe the allowable sequence of function calls, this section gives some rules regarding the maintenance of the state of the interface:

- It is the responsibility of the transport provider to keep record of the state of the interface as seen by the transport user.
- The transport provider will never process a function that places the interface out of state.
- If the user issues a function out of sequence, the transport provider will indicate this were possible through an error return on that function. The state will not change. In this case, if any data is passed with the function when not in the T_DATAXFER state, that data will not be accepted or forwarded by the transport provider.
- The uninitialized state (T_UNINIT) of a transport endpoint is the initial state, and the endpoint must be initialized and bound before the transport provider may view it as active.
- The uninitialized state is also the final state, and the transport endpoint must be viewed as unused by the transport provider. The *t_close*() function will close the transport provider and free the transport library resources for another endpoint.
- According to the state table in the section entitled State Tables, *t_close*() should only be issued from the T_UNBND state. If it is issued from any other state and no other user has that endpoint open, the action will be abortive, the transport endpoint will be successfully closed, and the library resources will be freed for another endpoint. When *t_close*() is issued, the transport provider must ensure that the address associated with the specified transport endpoint has been unbound from that endpoint. Also, the provider should send appropriate disconnects if *t_close*() is not issued from the unbound state.

The following rules apply only to the connection-mode transport service:

- The transport connection release phase can be initiated at any time during the connection establishment phase or data transfer phase.

- The only time the state of a transport service interface of a transport endpoint may be transferred to another transport endpoint is when the *t_accept*() function specifies such action. The following rules then apply to the cooperating transport endpoints:

  - The endpoint that is to accept the current state of the interface must be bound to an appropriate protocol address and must be in the T_IDLE state.

  - The user transferring the current state of an endpoint must have correct permissions for the use of the protocol address bound to the accepting transport endpoint.

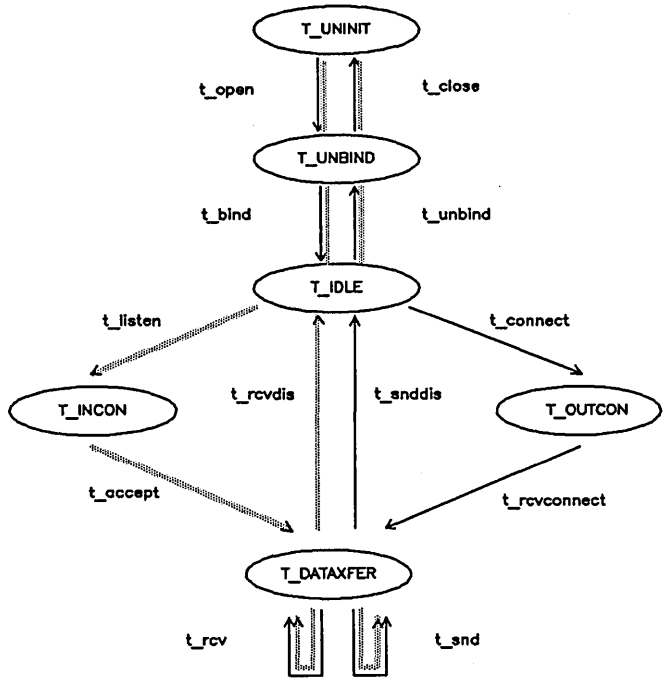
  - The endpoint that transfers the state of the transport interface is placed into the T_IDLE state by the transport provider after the completion of the transfer if there are no more outstanding connect indications.

Example in Connection-Oriented Mode

The following table shows the allowable sequence of functions of an active user and passive user communicating using a connection mode transport service. This example is not meant to show all the functions that must be called but rather to highlight the important functions that request a particular service. Blank lines are used to indicate that a function would be called by another user prior to a related function being called by the remote user. For example, the active user calls *t_connect*() to request a connection and the passive user would receive an indication of the connect request (using the return from *t_listen*()) and then would call the *t_accept*().

The state diagram that follows shows the flow of the events through the various states. The active user is represented by a solid line and the passive user is represented by a dashed line. This example shows a successful connection being established and terminated using connection-mode transport service without orderly release.

Active User	Passive User
t_open()	*t_open*()
t_bind()	*t_bind*()
	t_listen()
t_connect()	
	t_accept()
t_rcvconnect()	
t_snd()	
	t_rcv()
t_snddis()	
	t_rcvdis()
t_unbind()	*t_unbind*()
t_close()	*t_close*()

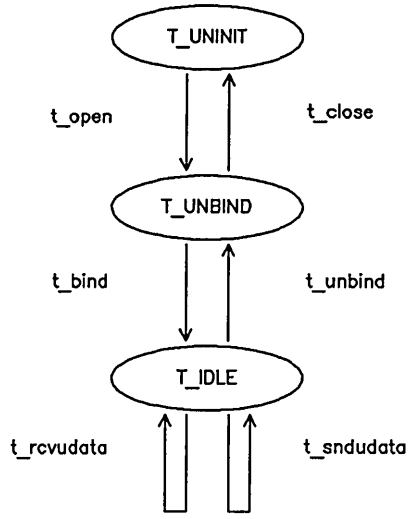


Example in Connectionless Mode

The following table shows the allowable sequence of functions of a user A and user B communicating using a connectionless transport service. This example is not meant to show all the functions that must be called but rather to highlight the important functions that request a particular service. Blank lines are used to indicate that a function would be called by another user prior to a related function being called by the remote user.

The state diagram that follows shows the flow of the events through the various states. This example shows a successful exchange of data between the user A and the user B.

User A	User B
t_open()	*t_open*()
t_bind()	*t_bind*()
t_sndudata()	
	t_rcvudata()
t_unbind()	*t_unbind*()
t_close()	*t_close*()



Writing Protocol Independent Software

In order to maximize portability of XTI applications between different kinds of machines and to support protocol independence, there are some general rules:

1. An application should only make use of the functions and mechanisms described as being mandatory features of XTI.

2. In the connection-mode service, the concept of a transport service data unit (TSDU) may not be supported by all transport providers. The user should make no assumptions about the preservation of logical data boundaries across a connection.

3. If an application is not intended to run only over an ISO transport provider, then the name of device should not be hard-coded into it. While software may be written for a particular class of service (for instance, connectionless-mode service), it should not be written to depend on any attribute of the underlying protocol.

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

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

6. Protocol-specific addressing issues should be hidden from the user program. Similarly, the user must have some way of accessing destination addresses in an invisible manner, such as through a name server. However, the details for doing so are outside the scope of this interface specification.

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

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

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

A

Error Codes

Error Codes

This appendix lists error codes returned by the Protocol Manager, the Link Layer, and XTI.

Protocol Manager Error Codes

Mnemonic	Code	Description
ErcInvalidSize	53312	An invalid size is specified in the request pb/cbs. This error code is returned to an application if it specifies an invalid request or response cb size.
ErcNoSpace	53313	This code is returned to an application making a register request (either a Link Layer or Link Client) if the maximum number of applications as specified in the installation have already been registered.
ErcBadDeinstall	53314	Protocol Manager is active, that is, some Link Layers or Link Clients are registered. This error code is returned to the Deinstall utility if some of the Link Layers or Link Clients are active (registered with Protocol Manager). Note that Protocol Manager deinstallation is not allowed in this case.

Mnemonic	Code	Description
ErcInvalidName	53315	The name specified in a request is invalid. This error is returned if the length of name or node name specified in a request is invalid. A name is invalid if it is more than 12 characters. A node name is invalid if it has more than 12 characters or has mismatched braces (right brace missing).
ErcDuplicateName	53316	The name specified in a register request is already registered with the Protocol Manager. This error will be returned to a Link Layer or Link Client when it makes a register request and the name specified in the request is already registered with Protocol Manager.
ErcBadRetSize	53317	The size of the returned information is not sufficient. Invalid value in response pb/cbs.
ErcInvalidSDF	53318	Invalid SD File format. The Protocol Manager returns this error if the Link Layer Name in the SD file (the first parameter) is invalid.
ErcInvalidUser	53319	Invalid User number on a Deregister request received from a Link Layer or Link Client (a different user number than the one used in the original Register request).
ErcNotRegistered	53320	The Link Layer is not installed on the system. Returned to a Link Client when it makes the RequestLinkLayer request or to a Link Layer making a Deregister or Update request.

Mnemonic	Code	Description
ErcDeinsCM	53321	Returned to a deinstall request when request was issued on a different workstation.
ErcDeinsOS	53322	Returned to a deinstall request when the program is executing on a single partition real-mode operating system.

Link Layer Error Codes

Mnemonic	Code	Description
ErcAlreadyOpen	31100	The station address is already in use.
ErcNoAddress	31101	The station address is not configured or there are no more free addresses (LSAPs).
ErcNotOpen	31102	An attempt was made to access an unopen station. This error code is returned if an operation other than OpenStation and DirectLink is issued before OpenStation.
ErcWrongUser	31103	A service was requested by someone other than the opening user.
ErcBadCommand	31104	Undefined command in DirectLink.
ErcNotSupported	31105	A defined command which is not supported by this Link Layer was used in DirectLink.
ErcXmtDataTrunc	31106	The buffer size given in WriteDLFrame request is larger than the maximum frame install parameter (if any).

Mnemonic	Code	Description
ErcRcvDataTrunc	31107	Received frame is larger than the maximum frame install parameter or buffer provided in ReadDLFrame request. The excess data is lost.
ErcNullBuffer	31108	The buffer size specified in Read or WriteDLFrame request is zero.
ErcLineDown	31109	The Line (media) is inoperable.
ErcAborted	31110	Abort or terminate condition.
ErcReqCanceled	31111	An outstanding request was canceled due to a DirectStation cancel command.
ErcDeinstOS	31112	Attempt to deinstall on a single-partition operating system or attempt to deinstall from a remote workstation.
ErcDeinstCM	31113	Deinstall request when the Context Manager is installed.
ErcLLActive	31114	Deinstall request when there is a Link Client or the Link Layer is active.
ErcBadRetSize	31125	The request code list or station information buffer is too small.
ErcBadLLName	58200	Duplicate or invalid Link Layer name.
ErcSwapped	58201	CM Swapping condition response.
ErcInvalidSDF	58203	The specified SDF file is either invalid or nonexistent.

Mnemonic	Code	Description
ErcInvalidSDStringFmt	58204	The passed SD string is invalid.
ErcInvalidSDContent	58205	The information in the SD string is not valid for this Link Layer.
ErcBufferTooSmall	58207	The buffer passed to the Link Layer is too small to use.
ErcDeinstallLink	58208	There has been a fatal error in the link. Deinstall the link and restart.
ErcBadLinkClientName	58209	The Link Client Name passed in the request was invalid.
ErcBadInstallParam	58211	Invalid parameter value specified during installation.
ErcBadDataSize	58212	The size of the data passed to the Link Layer is incorrect.
ErcGroupAddrDetect	58213	A message was received with a broadcast or multicast address.
ErcStnClosed	58217	A CloseStation request was received. This error code is returned with the read/write requests which are queued up in the Link Layer.
ErcInvalidState	58218	This error is returned by the link if it receives a request when it is in an invalid state. For example: A WriteDLFrame (I-Frame) command before a link connection is active.

Mnemonic	Code	Description
ErcReceiveTruncation	58219	This error code, like 31107 (ErcRcvDataTrunc), indicates that the received frame is larger than the buffer provided in ReadDLFrame. However, this error code indicates that the remaining data is still available and will be returned on the next ReadDLFrame.
ErcInvalidLSAPSpec	58224	Invalid remote LSAP specification for this Link Layer.
ErcLinkReset	60211	This error code indicates that an established link connection has been reset at Layer Two and is reestablished. All outstanding WriteDLFrame requests are returned with this status code if LAPD or the network resets the connection.
ErcInvalidFrameSpec	60212	Invalid frame type specification for this Link Layer.
ErcNoResources	60213	This error code indicates that the Link Layer has run out of some resource. When returned in response to a ReadDLFrame or WriteDLFrame request, it indicates that the client has issued too many of these requests simultaneously.
	60214	Reserved for future use by Link Layer API.
	60215	Reserved for future use by Link Layer API.
	60216	Reserved for future use by Link Layer API.

Mnemonic	Code	Description
ErcLinkDown	60217	This error code indicates that the link connection has been released or has not been established. This error is returned to the OpenStation request or DirectStation (bCommand = OpenLogicalLink or ResetLogicalLink). It indicates that the link layer could not establish or reset a link connection on behalf of the user and that the link connection is down. All outstanding WriteDLFrame requests are returned with this status code if the network releases the connection.

XTI Error Codes

The range 60113 to 60210 is reserved for the Transport Layer API.

Mnemonic	Code	Description
ErcBadAddress	60113	Corresponds to [TBADADDR].
ErcBadOption	60114	Corresponds to [TBADOPT].
ErcAccess	60115	Corresponds to [TACCES].
ErcBadfd	60116	Corresponds to [TBADF].
ErcNoAddress	60117	Corresponds to [TNOADDR].
ErcOutOfState	60118	Corresponds to [TOUTSTATE].
ErcInvalidSeqNum	60119	Corresponds to [TBADSEQ].

Mnemonic	Code	Description
ErcTLook	60121	Corresponds to [TLOOK].
ErcUDataTooLong	60122	Corresponds to [TBADDATA].
ErcBufferOverflow	60123	Corresponds to [TBUFOVFLW].
ErcFlowControl	60124	Corresponds to [TFLOW].
ErcNoData	60125	Corresponds to [TNODATA].
ErcNoDisID	60126	Corresponds to [TNODIS].
ErcNoUnitData	60127	Corresponds to [TNOUDERR].
ErcBadFlag	60128	Corresponds to [TBADFLAG].
ErcNoOrderlyRelease	60129	Corresponds to [TNOREL].
ErcNotSupported	60130	Corresponds to [TNOTSUPPORT].
ErcStateChange	60131	Corresponds to [TSTATECHNG].
ErcNoSuchStruct	60132	Corresponds to [TNOSTRUCTTYPE].
ErcBadName	60133	Corresponds to [TBADNAME].
ErcBadQueueLength	60134	Corresponds to [TBADQLEN].
ErcAddressBusy	60135	Corresponds to [TADDRBUSY].

NOTE: [TSYSERR] is returned by the XTI library whenever the response error code is outside the bounds 60113-60210.

B

Link API Event Codes

Event Codes

Event codes are returned by the ReadDLFrame request if needed by the application.

Catastrophic events (recommended action is to Deinstall the Link Layer or reboot):

03 Internal catastrophic event.

Critical events (reconfiguration recommended):

11 Buffer resources exhausted.

Other events (no action is required):

20 Line active subsequent to a line down event (33).

21 Write I-frame confirmed.

29 FRMR returned.

Operational events (operator intervention is or may be required):

33 Line or physical link down (examples are DSR off for a modem connection, or disconnection from the ring for Token Ring).

34 Link down due to security problem.

35 Logical Link up (examples: SABM, SNRM, SABME received while Link is down).

36 Logical Link down (example: DISC received while Link is up).

37 Logical Link reset (examples: SABM, SNRM, SABME received while Link up).

Operational events (programmatic intervention may be required):

40 Connect Indication

41 Disconnect Indication

42 Reset

43 Flow control (the byte following RemoteLSAPData is zero if the remote node has just sent its first RR following an RNR, and non-zero if the remote node has just sent its first RNR).

C

Transport API Definitions

XTI Library Error Codes

Mnemonic	Code	Description
TBADADDR	1	Incorrect address format
TBADOPT	2	Incorrect option format
TACCES	3	Incorrect permissions
TBADF	4	Illegal transport *fd*
TNOADDR	5	Couldn't allocate *addr*
TOUTSTATE	6	Out of state
TBADSEQ	7	Bad call sequence number
TSYSERR	8	System error
TLOOK	9	Event requires attention
TBADDATA	10	Illegal amount of data
TBUFOVFLW	11	Buffer not large enough
TFLOW	12	Flow control
TNODATA	13	No data
TNODIS	14	Discon_Ind not found on queue
TNOUDERR	15	Unit data error not found

TBADFLAG	16	Bad flags
TNOREL	17	No orderly release found on queue
TNOTSUPPORT	18	Primitive not supported
TSTATECHNG	19	State is in process of changing
TNOSTRUCTYPE	20	Unsupported struct-type requested
TBADNAME	21	Invalid transport provider name
TBADQLEN	22	Queue length is zero
TADDRBUSY	23	Address in use

XTI Library Event Codes

Mnemonic	Code	Description
T_LISTEN	0001h	Connection indication received
T_CONNECT	0002h	Connect confirmation received
T_DATA	0004h	Normal data received
T_EXDATA	0008	Expedited data received
T_DISCONNECT	0010h	Disconnect received
T_UDERR	0040h	Datagram error indication
T_ORDREL	0080	Orderly release indication
T_GODATA	0100	Sending normal data is again possible
T_GOEXDATA	0200	Sending expedited data is again possible
T_SENDSYNC	8000	The transport service provider requests this transport service client to send sync data, so that another user of this endpoint can be synchronized.

XTI Library Flag Definitions

Mnemonic	Code	Description
T_MORE	001h	More data
T_EXPEDITED	0021h	Expedited data
T_NEGOTIATE	004h	Set options
T_CHECK	008h	Check options
T_DEFAULT	010h	Get default options
T_SUCCESS	020h	Successful
T_FAILURE	040h	Failure

XTI Library Service Type Definitions

Mnemonic	Code	Description
T_COTS	01	Connection-oriented transport service
T_COTS_ORD	02	Connection-oriented with orderly release
T_CLTS	03	Connectionless transport service

XTI Library State Codes

Mnemonic	Code	Description
T_UNBIND	1	Unbound
T_IDLE	2	Idle
T_OUTCON	3	Outgoing connection pending
T_INCON	4	Incoming connection pending
T_DATAXFER	5	Data transfer

T_OUTREL	6	Outgoing release pending
T_INREL	7	Incoming release pending

XTI General Purpose Values

Mnemonic	Code
T_YES	1
T_NO	0
T_UNUSED	-1
T_NULL	0
T_ABSREQ	8000h

XTI Flags for t_open

Mnemonic	Code	Description
O_RDWR	0002h	Read/Write flag
O_NONBLOCK	0004h	If set, indicates that Non-Blocking mode is requested.
O_FILESPEC	2000h	If set, indicates that the pbProviderName parameter for *t_open* points to a Parameter Definition File, which contains the actual Provider Name and possible additional parameters.
O_NAME_PDS	4000h	If set, indicates that that pbProviderName parameter for *t_open* points to a string which contains both a Provider Name and a Parameter Definition String.

Specific ISO Option and Management Parameters

Mnemonic	Code
T_CLASS0	0
T_CLASS1	1
T_CLASS2	2
T_CLASS3	3
T_CLASS4	4
T_PRITOP	0
T_PRIHIGH	1
T_PRIMID	2
T_PRILOW	3
T_PRIDFLT	4
T_NOPROTECT	1
T_PASSIVEPROTECT	2
T_ACTIVEPROTECT	4
T_LTPDUDFLT	128

TCP Specific Environment

Mnemonic	Code
T_ROUTINE	0
T_PRIORITY	1
T_IMMEDIATE	2
T_FLASH	3
T_OVERRIDEFLASH	4
T_CRITIC_ECP	5
T_INETCONTROL	6
T_NETCONTROL	7

D

ISO Transport Protocol Information

This appendix describes the protocol-specific information that is relevant for ISO transport providers.

Generalities

Protocol Address

In an ISO environment, the protocol address is the T-Address (Transport Address). This contains the Transport Service Access Point (TSAP) and Network Service Access Point (NSAP) for both the local and the remote nodes.

Sending Zero Octets of Data

The transport service definition, both in connection-oriented mode and in connectionless mode, does not permit transmission of a TSDU of zero octets. So, if a *nbytes* or *len* parameter is set to zero, the call to *t_snd* or *t_sndudata* call will always return unsuccessfully with -1 and *t_errno* set to [TBADDATA].

Expedited Data

In connection-oriented mode and when the transport class permits it, expedited data option must be negotiated during the connection establishment phase. In connectionless mode, this feature is not supported.

Option Structure and Parameters

What follows is a description of the ISO options and options structure for both connection-mode and connectionless-mode transport.

Connection Mode

The functions *t_accept*, *t_listen*, *t_connect*, *t_rcvconnect*, and *t_optmgmt* contain an *opt* argument which is of type **struct netbuf**. The *opt.buf* argument of the **netbuf** structure should point to a **isoco_options** structure which contains the following parameters of quality of service:

Offset	Field	Size (Bytes)	Contents
0	throughput	32	**thrpt** structure containing throughput
32	transdel	16	**reqvalue** structure containing transit delay
48	reserrorrate	8	**rate** structure containing residual error rate
56	transffailprob	8	**rate** structure containing transfer failure probability
64	estfailprob	8	**rate** structure containing connection establishment failure probability
72	relfailprob	8	**rate** structure containing connection release failure probability
80	estdelay	8	**rate** structure containing connection establishment delay
88	reldelay	8	**rate** structure connection release delay
96	connresil	8	**netbuf** structure connection resilience
104	protection	2	**protection** valve
106	priority	2	**priority** valve
108	mngmt	13	**management** structure containing management parameters
121	expd	1	expedited data: T_YES or T_NO

Where the **rate** structure contains:

Offset	Field	Size (Bytes)	Contents
0	targetvalue	4	target value
4	minacceptvalue	4	minimum acceptable value

The **reqvalue** structure contains:

Offset	Field	Size (Bytes)	Contents
0	called	8	rate structure containing called rate
8	calling	8	rate structure containing calling rate

The **thrpt** structure contains:

Offset	Field	Size (Bytes)	Contents
0	maxthrpt	16	**reqvalue** structure containing maximum throughput
16	avgthrpt	16	**reqvalue** structure containing average throughput

The **management** structure contains:

Offset	Field	Size (Bytes)	Contents
0	dflt	2	T_YES: the following parameters are ignored: default values are used T_NO: the following parameters are used:
2	ltpdu	2	maximum length of TPDU (in octets)
4	reastime	2	reassignment time (in seconds)
6	class	1	preferred class; value: T_CLASS0-T_CLASS4
7	altclass	1	alternative class
8	extform	1	extended format: T_YES or T_NO
9	flowctrl	1	flow control: T_YES or T_NO
10	checksum	1	checksum (cl. 4): T_YES or T_NO
11	netexp	1	network expedited data: T_YES or T_NO
12	netrecptcf	1	receipt confirmation: T_YES or T_NO

General Remarks

- Unused fields (values or flags) should be set to T_UNUSED.
- If used, the flag fields (*extform, flowctrl, checksum, netexp, netrecptcf* and *expd*) will be set to either T_YES or T_NO.
- If the user does not want to define any option, the options pointer is set to the null pointer.
- When the variable *opt* is used as output parameter (on calling *t_listen*) the item *opt.buf* will point, before the call, to an initialized **isoco_options** structure.
- If the user does not want to specify some requirements concerning the QOS parameters (except *protection* and *priority*), the parameter must be set to T_UNUSED.

- In the **rate** structure, the two fields which contain the ratio of lost or erroneous TSDUs to total TSDUs transmitted are expresssed as a power of 10. The implicit length of TSDU is 128 octets.

- The **reqvalue** structure is used to describe the throughput and the transit delay. In the **reqvalue** structure, the *minacceptvalue* field of **struct rate** contains the minimum acceptable value for throughput and the maximum acceptable value for transit delay. These fields may be set to 0 if the throughput or transit delay are not absolute (mandatory) requirements for the user.

- For the *throughput* field, if *avgthrpt* (average throughput) is not defined (both fields set to T_UNUSED), XTI considers that the average throughput has the same values as the maximum throughput (*maxthrpt*).

- The *priority* field may be set by one of the five following symbolic constants to define the level of priority:

T_PRIDFLT	lower level (default level)
T_PRILOW	low level
T_PRIMID	medium level
T_PRIHIGH	high level
T_PRITOP	higher level of priority

  The *priority* field is not considered as a mandatory requirement. If the transport provider does not support this feature, it will ignore this user requirement.

- The *protection* field defines the general level of protection. Several levels are defined. The symbolic constants listed below are used as flags to specify the required level of protection:

T_NOPROTECT	no protection feature
T_PASSIVEPROTECT	protection against passive monitoring

T_ACTIVEPROTECT protection against modification, replay, deletion or addition

Both T_PASSIVEPROTECT and T_ACTIVEPROTECT may be set simultaneously but are exclusive with T_NOPROTECT. If the T_ACTIVEPROTECT or T_PASSIVEPROTECT flags are set, the user may indicate that this is an absolute requirement by also setting the T_ABSREQ flag. In this case, the function called will fail if the transport provider cannot respect this condition.

Connectionless Mode

In connectionless mode, the functions *t_sndudata*, *t_rcvudata* and *t_optmgmt* called to send or receive a data unit, or retrieve information, use a variable, *unitdata.opt*, as input or output parameter. This variable, whose type is **struct netbuf**, comprises a field, *opt.buf*, which must point to a **struct isocl_options**. This structure contains:

Offset	Field	Size (Bytes)	Contents
0	transdel	2	struct rate transit delay
2	reserrorrate	2	struct rate residual error rate
4	protection	2	
4	priority	2	

where **struct rate** is as defined for connection-oriented options.

General Remarks

- The value of the standard priority for the *priority* field is T_PRIDFLT. The definition of the *priority* field is the same as in connection-oriented mode.
- For the transit delay field (*transdel*), if described, the *minacceptvalue* field is considered as containing a mandatory requirement. Otherwise the user should set it to T_UNUSED.

- For the *reserrorrate* field, see the remarks for the same parameters in the section on Connection Mode.

- On the call to *t_rcvudata*, the variable *unitdata.opt* is used as an output parameter. As described in Connection Mode, the user must provide an initialized **isocl_options** structure.

- The transit delay parameter is the most important and useful in Connectionless mode.

Warnings on the Use of Option Parameters

Both sections above describe a mechanism for selecting and/or negotiating options in connection or connectionless mode through the Transport Service interface. It is important to note that:

- Some of the parameters listed are not well defined (by ISO) in terms of their use and interpretation; these parameters are:

  - protection

  - transfer failure probability

  - connection resilience

  Consequently, the use of these parameters is not recommended until their use is better defined.

- Management options may be selected by some other (local) means, and the list of management options may be extended to include other such options related to any particular Transport Service implementation.

- Ultimately, a provider may ignore all selections, except in the case where a mandatory selection cannot be supported; in this case, the provider will refuse the selection.

Class Limitation of Some Parameters

The fields listed below are significant only in the following cases:

expd	if *class* > 2 or class 2 with the *explicit flow control* option selected
protection	if *class* > 0
flowctrl	if *class* = T_CLASS2
checksum	if *class* = T_CLASS4
netexp	if *class* = 1
netrecptcf	if *class* = 1

Default Values

The following table provides the default values of some parameters when the corresponding fields are set to T_UNUSED or, for all except the first two, when the *dflt* field in the structure **management** is set to T_YES by the user.

Field Name	Default Value	Meaning
expd	T_NO	no expedited flow
priority	T_PRIDFLT	standard priority
checksum	T_NO	no checksum
extform	T_NO	no extended format
flowctrl	T_YES	flow control
netrcptcf	T_NO	no receipt confirmation
netexp	T_NO	no network expedited data
ltpdu	T_LTPDUDFLT	length of TPDU

The fields *class*, *altclass* and *reastime* are system and connection dependent.

Functions

t_accept

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 have been bound to the same address as *fd*, with the *qlen* parameter set to 0 when the *t_bind* was called for that *resfd*.

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, **ISO 8072, Transport Service Definition** does not authorize acceptance of an incoming connection indication with a responding address different from the called address except under certain conditions (see **IS 8047, Paragraph 12.2.4, Responding address**), but it also states this may be changed in the future.

t_bind

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

t_connect

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.

If the user chooses to negotiate options, the *sndcall.opt* structure must point to the **isoco_options** structure. The setting of the *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.

t_listen

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*), 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. Thus, if the user enqueues the connect indications for too long a time before responding to them, the transport provider initiating the connection will disconnect it.

t_open

The function *t_open* is called at the first step in the initialization 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 should be the values returned by the call to *t_open* with an ISO transport provider:

Parameters	Before Call	After Call	
		Connection	Connectionless
name	x	/	/
oflag	x	/	/
info.addr	/	x	x
info.options	/	x	x
info.tsdu	/	-1	-1
info.etsdu	/	16	-2
info.connect	/	32	-2
info.discon	/	64	-2
info.servtype	/	T_COTS	T_CLTS

t_rcvconnect

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*.

t_rcvdis

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*.

t_rcvudata

The *unitdata.addr* structure specifies the remote TSAP. The quality of service associated with the received data unit is returned in the **isocl_options** structure pointed to by *unitdata.opt.buf*. If the T_MORE flag is set, an additional call to *t_rcvudata* is needed to retrieve the entire TSDU. Only normal data is returned via a call to *t_rcvudata*.

t_rcvuderr

The *uderr.addr* structure contains the remote TSAP. The **isocl_options** structure pointed to by *uderr.opt.buf* identifies the quality of service associated with the data unit received.

t_snddis

Since at most 64 octets of data may be sent with the disconnect, *call.udata.len* will have a value less than or equal to 64.

t_sndudata

The *unitdata.addr* structure specifies the remote TSAP. If the user chooses to associate quality of service with this request, the *unitdata.opt* structure must point to the **isocl_options** structure. An ISO connectionless transport service does not support the sending of expedited data.

E

Internet Transport Specific Information

This appendix describes the protocol-specific information that is relevant for TCP and UDP transport providers.

Generalities

- T_MORE flag and TSDUs

  The notion of TSDU is not supported by a TCP transport provider, thus the T_MORE flag shall be ignored when TCP is used. The TCP PUSH flag cannot be used through the XTI interface because **Section 9.2.7** from **[ref 3]** states that: "Successive pushes may not be preserved because two or more units of pushed data may be joined into a single pushed unit by either the sending or receiving TCP. Pushes are not visible to the receiving Upper Level Protocol and are not intended to serve as a record boundary marker."

- Expedited data

  Normal and expedited flows are not two distinct flows in TCP. Once the send window is filled, the local process is not allowed to send any data, normal or expedited. When the send window opens again, it is open for both normal and expedited data. Expedited data cannot be sent by UDP.

- Orderly release

  The orderly release functions, *t_sndrel* and *t_rcvrel*, were defined to support the orderly release facility of TCP. However, its use is not recommended so that applications using TCP may be ported to use an ISO transport provider. The specification of TCP states that only established connections may be closed with orderly release, (such as on an endpoint in T_DATAXFER state).

- Timeouts

  It is not possible to redefine a new value for the timeout on each TCP request. The user can specify a value at the time of connection establishment which is fixed for the life of the connection.

Option Parameters

What follows is a description of the protocol-specific transport options for TCP and UDP.

Connection Mode: TCP

The functions *t_accept*, *t_listen*, *t_connect*, *t_rcvconnect*, and *t_optmgmt* contain an *opt* argument which is of type **struct netbuf**. The *opt.buf* argument of the **netbuf** structure should point to a **tcp_options** structure which contains:

Offset	Field	Size (Bytes)	Contents
0	precedence	2	
2	timeout	4	abort timeout (expressed in milliseconds for TCP)
8	max_seg_size	4	maximum segment size
12	secopt	10	**secoptions** structure containing security options for TCP

where the **secoptions** structure contains:

Offset	Field	Size (Bytes)	Contents
0	security	2	security field
2	compartmente	2	compartment
4	handling	2	handling restrictions
6	tcc	4	transmission control code

General Remarks

- Unused fields (values or flags) should be set to T_UNUSED.
- If the user does not want to define any option, the options pointer is set to the null pointer.
- When the variable *opt* is used as output parameter (on calling *t_listen*) the item *opt.buf* will point, before the call, to an initialized **tcp_options** structure.
- If the user does not want to specify some requirements, the corresponding parameters must be set to T_UNUSED.
- The *precedence* field is used to express the precedence level in TCP. It may be set to one of the following symbolic constants:

  T_ROUTINE

  T_PRIORITY

  T_IMMEDIATE

  T_FLASH

  T_OVERRIDEFLASH

  T_CRITIC_ECP

T_INETCONTROL

T_NETCONTROL

- The four parameters of the **secoptions** structure are provided to define the security options:

  1. The *security* field defines the general level of security.

  2. The *compartment* field defines the compartment parameter.

  3. The *handling* field defines the handling restrictions parameter.

  4. For the *tcc* field which defines the transmission control code, only the 3 low-order bytes are used.

Connectionless Mode: UDP

UDP uses no options, so the *opt* arguments to *t_snddata* and *t_rcvudata* should be set to zero or the null pointer, as appropriate.

Functions

t_accept

Since data may not be sent with a connect accept, *call.udata.len* must be set to zero.

t_bind

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

t_connect

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

If the user chooses to negotiate options, the *sndcall.opt* structure must point to the **tcp_options** structure. Since data may not be sent with a *t_connect*, *sndcall.udata.len* must be set to zero.

t_listen

Since data may not be sent with a connect, *call.udata.maxlen* must be set to zero before the call to *t_listen*. The *call.addr* structure contains the remote calling socket.

t_open

The function *t_open* is called at the first step in the initialization 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 should be the values returned by the call to *t_open* and *t_getinfo* with the indicated transport providers:

Parameters	Before Call	After Call	
		TCP	UDP
name	x	/	/
oflag	x	/	/
info.addr	/	x	x
info.options	/	x	-2
info.tsdu	/	0	x
info.etsdu	/	-1	-2
info.connect	/	-2	-2
info.discon	/	-2	-2
info.servtype	/	T_COTS or T_COTS_ORD	T_CLTS

t_rcv

The T_MORE flag should be ignored. It T_EXPEDITED flag is set, out-of-band data is received.

t_rcvconnect

Since data may not be sent with a connect, *call.udata.maxlen* must be set to zero before the call to *t_rcvconnect*. On return, the *call.addr* structure contains the remote calling socket.

t_rcvdis

Since data may not be sent with a disconnect, the *discon.udata* structure will not be meaningful.

t_rcvudata

No options are supported by UDP, so *unitdata.opt.maxlen* should be zero.

t_snd

The T_MORE flag should be ignored. If T_EXPEDITED flag is set, out-of-band data is sent (at least one octet must be sent).

t_snddis

Since data may not be sent with a disconnect, *discon.udata.len* must be set to zero.

t_sndudata

No options are supported by UDP, so *unitdata.opt.len* should be zero. Also, be aware that the maximum size of a connectionless TSDU varies among implementations.

Index

Accepting a connect request, 4-47
Activating a transport endpoint, 4-53
API (Application Programming Interface)
  changes to specification, 1-6
  purpose of, 1-1
Application Layer, definition of, 1-3
Application Programming Interface. *See* API
Applications, CTOS
  networking services, 1-5
  portability of, 1-1
  porting, 1-5
ARPANET, 4-2
Asynchronous events, polling for, 4-85
Asynchronous mode
  establishing connection, 4-106
  transport service, 4-10

bCommand values, 3-29, 3-33, 3-35, 3-36
bind function description, 4-53 to 4-59
Binding a transport endpoint, 4-53
Blocks
  Generic Statistical Header. *See* GHB
  Service Client Descriptor Block. *See* SCDBB
  Service Provider Descriptor. *See* SPD

Changes to API specification, 1-6
Channel fields, 2-21
Client. *See* Service client
Close Logical Link command, 3-32
CloseStation request syntax description, 3-11 to 3-12
Closing
  link connections, 3-11
  transpoint endpoint *fd*, 4-60
Colons in parameter definition file, 2-2
Commands
  Close Logical Link, 3-32
  Connect Logical Link , 3-34
  Link Layer dependent, 3-33, 3-36
  Link Layer independent, 3-35, 3-29
  Open Logical Link, 3-31
  in parameter definition file, 2-2
  Query Station, 3-30
Communication path, 4-6
Connect Logical Link command, 3-34
Connect requests, accepting, 4-47
Connecting
  with Link Layer, 3-3
  transport users, 4-64
Connection mode
  connection/release/data transfer state table, 4-39
  example, 4-156
  option structure, 4-42
  TCP options, E-2 to E-4
  transport service, 4-8

Connectionless mode
data transfer state table, 4-38
example, 4-158
option structure, 4-42
transport service, 4-8
UDP options, E-4 to E-6
Connections
acknowledging release of, 4-116
asychronous mode, 4-106
closing, 3-11
detecting request for, 4-80
establishment, 4-16 to 4-18
multiple, 4-10
rejecting request for, 4-134
releasing transport, 4-138
CTOS applications. *See* Applications, CTOS
CTOS/Open Advisory Council, 1-1

Data
receiving, 4-19 to 4-20, 4-102, 4-119
sending, 4-21 to 4-22, 4-128, 4-141
Data Link Layer, definition of, 1-2
Data structures
Link Layer, 3-27 to 3-28
Protocol Manager, 2-21 to 2-22
synchronizing, 4-147
remote LSAP, 3-31
Declaring
Service Client name, 2-8
Service Provider name, 2-3
Deinstalling
Service Client, 2-11
Service Provider, 2-6
DeregisterServiceClient syntax description, 2-11 to 2-12
DeregisterServiceProvider syntax description, 2-6 to 2-7
Device independence, 3-2
Device specification of Service Provider, 2-13
DirectLink commands, 3-35 to 3-36
DirectLink request syntax description, 3-24 to 3-26
DirectStation Commands, 3-29 to 3-34
DirectStation request syntax description, 3-21 to 3-23
Disabling transport endpoints, 4-152
Disconnect, cause of, 4-111
Documents, 4-5

EM. *See* Event Management
Error codes
Link Layer, A-3 to A-7
Protocol Manager, A-1 to A-3
XTI Library, C-1 to C-2
Error handling, 4-9
Errors, receiving information about, 4-125
ETSDU (Expedited Transport Service Data Unit), 4-20
Event codes, XTI Library, C-2
Event Management (EM), 4-12 to 4-14
Events
incoming, 4-35 to 4-36
outgoing, 4-33 to 4-34
at transport endpoint, 4-12
Examples
connectionless mode, 4-158
connection-oriented mode, 4-156
Expedited Transport Service Data Unit. *See* ETSDU

Failures, 4-9
fd (file descriptor), 4-6
File descriptor. See *fd*
File, rqLabl.asm, 3-4
Flag definitions, XTI Library, C-3 to C-3
Frames
requesting, 3-13
writing, 3-17
Functions calls, sequence of, 4-155

GHB (Generic Statistical Header Block), 3-27

Initialization/deinitialization state table, 4-38
Initializing transport endpoint, 4-89
International Standards Organization. *See* ISO
Internet transport information, E-1 to E-6
IP Layer, 3-1
ISO (International Standards Organization)
  management parameters, C-5
  protocols, 4-2
  transport protocol, D-1 to D-11

Layer 1. *See* Physical Layer
Layer 2. *See* Data Link Layer
Layer 3. *See* Network Layer
Layer 4. *See* Transport Layer
Layer 5. *See* Session Layer
Layer 6. *See* Presentation Layer
Layer 7. *See* Application Layer
Layers of OSI Reference Model, 1-2 to 1-4
Length of Link Layer Name, 3-4
Library procedural interface, 4-1
Link API event codes, B-1 to B-2
Link Clients, 3-2
  connecting with Link Layer, 3-3
  opening LSAP for, 3-5, 3-8
Link Layer, 3-1 to 3-36
  data structures, 3-27 to 3-28
  dependent commands, 3-33, 3-36
  error codes, A-3 to A-7
  illustration of, 3-2
  independent commands, 3-29, 3-35
  name, 3-4
  operation, 3-24
  overview, 3-1
  PDF, 3-3
  requesting a service, 3-24
  service providers, 3-2
  services, 3-4 to 3-26
  Service Access Point. *See* LSAP
  station operation, 3-21
Listener application incoming *fds*, 4-7
Listening for a connect request, 4-80
Local management functions, 4-15
Long-lived interactions, 4-8
Long-lived Station Handle, 3-5
LSAP (Link Layer Service Access Point)
  opening for Link Client, 3-5, 3-8
  station requests, 3-2

Management functions, 3-35
Mandatory XTI features, 4-27
Messagees, transmitting from station, 3-17
Modes, service, 4-8. *See also* Connection mode; Connectionless mode

Names, Link Layer, 3-4
Netbeui protocol, 4-2
Network Layer, definition of, 1-2
Networking Services, CTOS/Open, 1-5

Object module procedures, 4-1
Open Logical Link command, 3-31
Open Systems Interconnection Reference Model. *See* OSI Reference Model
Opening
  LSAP for Link Client, 3-5, 3-8
  transport endpoint, 4-89

OpenStationLL request syntax description, 3-5 to 3-7
OpenStationSL request syntax description, 3-8 to 3-10
Operating system errors, 4-9
Optional XTI features, 4-28
Options , protocol-specific, 4-41
OSI Network Layer, 3-1
OSI Reference Model
definition, 1-2 to 1-4
illustration of, 1-4
O_NONBLOCK flag, 4-10

Parameter arrays, key to, 4-46
Parameter Definition Files. *See* PDF (Parameter Definition File)
PDF (Parameter Definition File)
for Link Layers, 3-3
purpose of, 2-1 to 2-2
Transport Layer, 4-45
Peer to peer protocols, 1-2
Physical Layer, definition of, 1-2
Pointers, unused, 3-34
Polling for asynchronous events, 4-85
Portability
of CTOS applications, 1-1
of networking applications, 1-5
rules, 4-159
Porting CTOS applications, 1-5
Presentation Layer, definition of, 1-3
Procedures
multiple XTI, 4-44
optional XTI, 4-43
Processes sharing *fds*, 4-7
Protocol addresses, endpoints on same, 4-8
Protocol Manager, 2-1 to 2-22
data structures, 2-21 to
error codes, A-1 to A-3
overview, 2-1
services, 2-2 to 2-22
updating, 2-19
Protocols
independent software, 4-159
negotiating options, 4-97
peer to peer, 1-2

Query Station command, 3-30
QueryProtocolManager syntax description, 2-16 to 2-18

rcvdis function description, 4-111 to 4-115
ReadDLFrame request
event codes, B-1 to B-2
syntax description, 3-13 to 3-16
Receiving data, 4-19 to 4-20, 4-102, 4-119
References, 4-5
RegisterServiceClient syntax description, 2-8 to 2-10
RegisterServiceProvider syntax description, 2-3 to 2-5
Rejecting a connect request, 4-134
Releasing
LSAP resources, 3-11
transport connection, 4-138
Remote LSAP structure, 3-31
Requesting
connection to transport user, 4-63
frames, 3-13
Link Layer service, 3-24
Requests, Protocol Manager, 2-2 to 2-20
RequestServiceProvider syntax description, 2-13 to 2-15
rqLabl.asm file, 3-4
Rules
connection mode transport service, 4-156
portability, 4-159
Transport Layer, 4-155

SCDB (Service Client Descriptor Block) structure, 2-22
SDF (Station Descriptor File), 3-4
Sending data, 4-21 to 4-22, 4-128, 4-141
Service Client. *See also* User
deinstalling, 2-11
name declaration, 2-8

Service Client. *See also* User **(cont.)**
new open request, 2-19
upper layer as, 1-4
Service Client Descriptor Block. *See* SCDB
Service Provider
change notification, 2-19
deinstalling, 2-6
device specification, 2-13
lower layer as, 1-4
name declaration, 2-3
status information, 2-16
Service Provider Descriptor Block. *See* SPDB
Service type definitions, XTI Library, C-3
Services, Link Layer, 3-4 to 3-26
Session Layer, 1-3
Short-lived Station Handle, 3-8
Short-term interactions, 4-8
SNA Path Control, 3-1
SPDB (Service Provider Descriptor Block) structure, 2-21
State codes, XTI Library, C-3
States
transition of, 4-37
of transport endpoint, 4-76
transport interface, 4-31 to 4-32
Station Descriptor File. *See* SDF
Station Handles
long-lived, 3-5
short-lived, 3-8
Station operation of Link Layer, 3-21
Station requests, 3-2
Stations, closing, 3-11
Status information, Service Provider, 2-16
Structures. *See* Data structures
Synchronizing data structures, 4-147
Synchronous mode transport service, 4-10

TCP
connection mode options, E-2 to E-4
environment mnemonics, C-6
TCP/IP, 4-2
TLOOK error returned by XTI calls, 4-40
Transmit buffer, 3-34
Transport API definitions, C-1 to C-6
Transport endpoint, 4-6
activating, 4-53
closing endpoint *fd*, 4-60
current event on, 4-85
current state, 4-76
disabling, 4-152
events, 4-12
initializing, 4-89
multiple events, 4-12
Transport Layer, 4-1 to to 4-160
definition of, 1-3
illustration of, 4-3
overview, 4-1
parameter definition file, 4-45
using the interface, 4-155 to
Transport protocol characteristics returned, 4-70
Transport provider states,4-31
Transport providers, 4-6
TSYSERR, 4-9
t_accept function description, 4-47 to 4-52
t_close function description, 4-60 to 4-62
t_connect function description, 4-63 to 4-69
t_errno, 4-9
t_getinfo function description, 4-70 to 4-75
t_getstate function description, 4-76 to 4-79
t_listen function description, 4-80 to 4-84
t_look function description, 4-85 to 4-88
t_open function
description, 4-89 to 4-96
XTI flags, C-4
t_optmgmt function description, 4-97 to 4-101
t_rcv function description, 4-102 to 4-105

t_rcvconnect function description, 4-106 to 4-110
t_rcvrel function description, 4-116 to 4-118
t_rcvudata function description, 4-119 to 4-124
t_rcvuderr function description, 4-125 to 4-127
t_snd function description, 4-128 to 4-133
t_snddis function description, 4-134 to 4-137
t_sndrel function description, 4-138 to 4-140
t_sndudata function description, 4-141 to 4-146
t_sync function description, 4-147 to 4-151
t_unbind function description, 4-152 to 4-154

UDP connectionless mode options, E-4 to E-6
UpdateProtocolManager syntax description, 2-19 to 2-20
User actions transport interface, 4-37

WriteDLFrame request syntax description, 3-17 to 3-19

X.25 Packet Layer, 3-1
XNS, 4-2
XTI
  connection establishment, 4-16 to 4-18
  connection-oriented mode, 4-14 to 4-15
  data transfer, 4-19 to 4-22
  error codes, A-7 to A-8
  flags for *t_open*, C-4
  function classifications, 4-30
  general purpose values, C-4
  initialization/deinitialization phase, 4-15 to 4-16
  mandatory features, 4-27
  optional features, 4-28
  optional procedures, 4-43
XTI Library
  error codes, C-1 to C-2
  event codes, C-2
  flag definitions, C-3 to C-3
  service type definitions, C-3
  state codes, C-3

CTOS *open*

Application Programming Interface Specification

Networking Services

With nearly one million workstation users worldwide, CTOS® provides an excellent platform for developing distributed, networked applications. CTOS is the only message-based, distributed operating system available for microprocessor-based workstations.

The CTOS/*Open* Networking Services consist of networking procedures that are common to all CTOS-based operating systems. By using these procedures, CTOS programs can directly communicate with workstations and mainframes to provide optimal networked solutions for today's business needs.

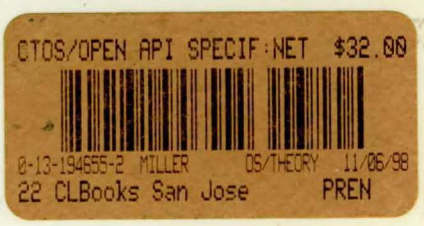
The transport layer of the CTOS/*Open* Networking Services complies with broadly accepted standards supported by IEEE, OSI, and X/Open. Programs that use this layer can remain portable across a wide range of platforms.

CTOS/Open Application Programming Interface Specification: Networking Services provides a clear and thorough description of each procedure in the CTOS/*Open* Networking Services. This book serves as an ideal reference for programmers who write networked applications.

In these pages, you'll find:

- An introduction of the CTOS/*Open* Standard for Networking Services
- A discussion of the seven layers of the OSI model
- Descriptions of the procedures for the CTOS/*Open* Protocol Manager
- Descriptions of the procedures for the CTOS/*Open* Link Layer Interface
- Descriptions of the procedures for the CTOS/*Open* Transport Layer Interface
- Descriptions of the error codes for the CTOS/*Open* Networking Services, as well as the event codes for the Link Layer Interface and the Transport Layer Interface

ISBN 0-13-194655-2



PRENTICE HALL
Englewood Cliffs, NJ 07632