

X/Open CAE Specification

Networking Services, Issue 4

X/Open Company Ltd.



© September 1994, X/Open Company Limited

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

Portions of this document are extracted from IEEE Std 1003.1-1990, copyright © 1990 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

Portions of this document were extracted from IEEE Draft Standard P1003.2/D12, copyright © 1992 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE. No further reproduction of this material is permitted without the written permission of the publisher. IEEE Std 1003.2-1992, copyright © 1992 by the Institute of Electrical and Electronics Engineers, Inc., and ISO/IEC 9945-2:1993, Information Technology — Portable Operating System (POSIX) — Part 2: Shell and Utilities, are technically identical to IEEE Draft Standard P1003.2/D12 in these areas.

Portions of this document are derived from copyrighted material owned by Hewlett-Packard Company, International Business Machines Corporation, Novell Inc., The Open Software Foundation, and Sun Microsystems, Inc.

X/Open CAE Specification

Networking Services, Issue 4

ISBN: 1-85912-049-0

X/Open Document Number: C438

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

Chapter 1	Common Information.....	1
1.1	Terminology.....	1
1.1.1	Shaded Text.....	2
1.2	Use and Implementation of Interfaces.....	2
1.2.1	C Language Definition.....	3
1.3	The Compilation Environment.....	4
1.3.1	X/Open UNIX Extension.....	4
1.3.2	The X/Open Name Space.....	5
1.4	Relationship to the XSH Specification.....	7
1.4.1	Error Numbers.....	7
1.5	Relationship to Emerging Formal Standards.....	7
Chapter 2	General Introduction to the XTI.....	9
Chapter 3	Explanatory Notes for XTI.....	11
3.1	Transport Endpoints.....	11
3.2	Transport Providers.....	11
3.3	Association of a UNIX Process to an Endpoint.....	12
3.4	Use of the Same Protocol Address.....	12
3.5	Modes of Service.....	13
3.6	Error Handling.....	13
3.7	Synchronous and Asynchronous Execution Modes.....	14
3.8	Event Management.....	16
Chapter 4	XTI Overview.....	17
4.1	Overview of Connection-oriented Mode.....	17
4.1.1	Initialisation/De-initialisation Phase.....	18
4.1.2	Overview of Connection Establishment.....	19
4.1.3	Overview of Data Transfer.....	20
4.1.4	Overview of Connection Release.....	22
4.2	Overview of Connectionless Mode.....	23
4.2.1	Initialisation/De-initialisation Phase.....	23
4.2.2	Overview of Data Transfer.....	23
4.3	XTI Features.....	25
4.3.1	XTI Functions versus Protocols.....	26
Chapter 5	States and Events in XTI.....	27
5.1	Transport Interfaces States.....	28
5.2	Outgoing Events.....	29
5.3	Incoming Events.....	30
5.4	Transport User Actions.....	31
5.5	State Tables.....	32

5.6	Events and TLOOK Error Indication.....	34
Chapter 6	The Use of Options in XTI	35
6.1	Generalities	35
6.2	The Format of Options.....	36
6.3	The Elements of Negotiation.....	37
6.3.1	Multiple Options and Options Levels.....	37
6.3.2	Illegal Options	37
6.3.3	Initiating an Option Negotiation.....	38
6.3.4	Responding to a Negotiation Proposal	39
6.3.5	Retrieving Information about Options.....	40
6.3.6	Privileged and Read-only Options.....	41
6.4	Option Management of a Transport Endpoint	42
6.5	Supplements	44
6.5.1	The Option Value T_UNSPEC	44
6.5.2	The info Argument	44
6.5.3	Summary	45
6.6	Portability Aspects.....	46
Chapter 7	XTI Library Functions and Parameters	47
7.1	How to Prepare XTI Applications.....	47
7.2	Key for Parameter Arrays	47
7.3	Return of TLOOK Error.....	47
	<i>t_accept()</i>	48
	<i>t_alloc()</i>	51
	<i>t_bind()</i>	53
	<i>t_close()</i>	56
	<i>t_connect()</i>	57
	<i>t_error()</i>	60
	<i>t_free()</i>	61
	<i>t_getinfo()</i>	63
	<i>t_getprotaddr()</i>	66
	<i>t_getstate()</i>	68
	<i>t_listen()</i>	69
	<i>t_look()</i>	71
	<i>t_open()</i>	73
	<i>t_optmgmt()</i>	76
	<i>t_rcv()</i>	82
	<i>t_rcvconnect()</i>	84
	<i>t_rcvdis()</i>	86
	<i>t_rcvrel()</i>	88
	<i>t_rcvudata()</i>	89
	<i>t_rcvuderr()</i>	91
	<i>t_snd()</i>	93
	<i>t_snddis()</i>	96
	<i>t_sndrel()</i>	98
	<i>t_sndudata()</i>	99
	<i>t_strerror()</i>	101

		<i>t_sync()</i>	102
		<i>t_unbind()</i>	104
Chapter	8	Sockets Interfaces	105
	8.1	Sockets Overview	106
		<i>accept()</i>	107
		<i>bind()</i>	109
		<i>close()</i>	111
		<i>connect()</i>	112
		<i>fcntl()</i>	115
		<i>fgetpos()</i>	116
		<i>fsetpos()</i>	117
		<i>ftell()</i>	118
		<i>getpeername()</i>	119
		<i>getsockname()</i>	120
		<i>getsockopt()</i>	121
		<i>listen()</i>	123
		<i>lseek()</i>	124
		<i>poll()</i>	125
		<i>read()</i>	126
		<i>recv()</i>	127
		<i>recvfrom()</i>	129
		<i>recvmsg()</i>	132
		<i>select()</i>	135
		<i>send()</i>	136
		<i>sendmsg()</i>	138
		<i>sendto()</i>	141
		<i>setsockopt()</i>	144
		<i>shutdown()</i>	146
		<i>socket()</i>	147
		<i>socketpair()</i>	149
		<i>write()</i>	151
Chapter	9	Sockets Headers.....	153
		<fcntl.h>	154
		<sys/socket.h>.....	155
		<sys/stat.h>	158
		<sys/un.h>.....	159
Chapter	10	IP Address Resolution Interfaces.....	161
		<i>endhostent()</i>	162
		<i>endnetent()</i>	164
		<i>endprotoent()</i>	165
		<i>endservent()</i>	166
		<i>gethostbyaddr()</i>	168
		<i>gethostname()</i>	169
		<i>getnetbyaddr()</i>	170
		<i>getprotobynumber()</i>	171

	<i>getservbyport()</i>	172
	<i>h_errno</i>	173
	<i>htonl()</i>	174
	<i>inet_addr()</i>	175
	<i>ntohl()</i>	177
	<i>sethostent()</i>	178
	<i>setnetent()</i>	179
	<i>setprotoent()</i>	180
	<i>setservent()</i>	181
Chapter 11	IP Address Resolution Headers	183
	<arpa/inet.h>.....	184
	<netdb.h>.....	185
	<netinet/in.h>.....	187
	<unistd.h>.....	188
Appendix A	ISO Transport Protocol Information.....	189
A.1	General.....	189
A.2	Options.....	190
A.2.1	Connection-mode Service	190
A.2.1.1	Options for Quality of Service and Expedited Data.....	190
A.2.1.2	Management Options	192
A.2.2	Connectionless-mode Service	194
A.2.2.1	Options for Quality of Service	194
A.2.2.2	Management Options	195
A.3	Functions	196
Appendix B	Internet Protocol-specific Information.....	199
B.1	General.....	199
B.2	Options.....	200
B.2.1	TCP-level Options	200
B.2.2	UDP-level Options.....	201
B.2.3	IP-level Options.....	202
B.3	Functions	205
Appendix C	Guidelines for Use of XTI.....	207
C.1	Transport Service Interface Sequence of Functions.....	207
C.2	Example in Connection-oriented Mode.....	208
C.3	Example in Connectionless Mode.....	210
C.4	Writing Protocol-independent Software.....	211
C.5	Event Management.....	212
C.5.1	Short-term Solution	212
C.5.2	XTI Events	213
C.6	The Poll Function	214
	<i>poll()</i>	215
C.7	Use of Poll.....	217
C.8	The Select Function.....	226
	<i>select()</i>	227

C.9	Use of Select	229
Appendix D	Use of XTI to Access NetBIOS.....	239
D.1	Introduction	239
D.2	Objectives	239
D.3	Scope.....	240
D.4	Issues	241
D.5	NetBIOS Names and Addresses.....	242
D.6	NetBIOS Connection Release	243
D.7	Options.....	244
D.8	XTI Functions.....	245
Appendix E	XTI and TLI.....	251
E.1	Restrictions Concerning the Use of XTI.....	251
E.2	Relationship between XTI and TLI	252
Appendix F	Headers and Definitions for XTI.....	253
Appendix G	Abbreviations.....	265
Appendix H	Minimum OSI Functionality (Preliminary Specification).....	267
H.1	General.....	267
H.1.1	Rationale for using XTI-mOSI.....	267
H.1.2	Migrant Applications	267
H.1.3	OSI Functionality	267
H.1.4	mOSI API versus XAP	268
H.1.5	Upper Layers Functionality Exposed via mOSI.....	268
H.1.5.1	Naming and Addressing Information used by mOSI.....	268
H.1.5.2	XTI Options Specific to mOSI	268
H.2	Options.....	270
H.2.1	ACSE/Presentation Connection-oriented Service.....	270
H.2.2	ACSE/Presentation Connectionless Service.....	271
H.2.3	Transport Service Options	272
H.3	Functions	273
H.4	Implementors) Notes	277
H.4.1	Upper Layers FUs, Versions and Protocol Mechanisms.....	277
H.4.2	Mandatory and Optional Parameters.....	277
H.4.3	Mapping XTI Functions to ACSE/Presentation Services.....	278
H.4.3.1	Connection-oriented Services	278
H.4.3.2	Connectionless Services	282
H.5	Complements to <xti.h>.....	283
H.6	XTI mOSI CR	287
Appendix I	SNA Transport Provider.....	293
I.1	Introduction	293
I.2	SNA Transport Protocol Information	294
I.2.1	General.....	294
I.2.2	SNA Addresses	295

I.2.3	Options.....	296
I.2.3.1	Connection-Mode Service Options.....	296
I.2.4	Functions	298
I.3	Mapping XTI to SNA Transport Provider	301
I.3.1	General Guidelines.....	302
I.3.2	Flows Illustrating Full Duplex Mapping	303
I.3.3	Full Duplex Mapping.....	312
I.3.3.1	Parameter Mappings.....	314
I.3.4	Half Duplex Mapping.....	324
I.3.5	Return Code to Event Mapping.....	325
I.4	XTI SNA CR.....	326

Appendix J The Internet Protocols..... 327

Glossary 329

Index..... 333

List of Figures

I-1	Active Connection Establishment, Blocking Version (1 of 2).....	303
I-2	Active Connection Establishment, Non-blocking Version (2 of 2).....	304
I-3	Passive Connection Establishment, Instantiation Version (1 of 3).....	305
I-4	Passive Connection Establishment, Blocking Version (2 of 3).....	306
I-5	Passive Connection Establishment, Non-blocking Version (3 of 3)....	307
I-6	XTI Function to LU 6.2 Verb Mapping: Blocking t_snd.....	308
I-7	XTI Function to LU 6.2 Verb Mapping: Non-blocking t_snd.....	309
I-8	XTI Function to LU 6.2 Verb Mapping: Blocking t_rcv.....	310
I-9	Mapping from XTI Calls to LU 6.2 Verbs (Passive side).....	311

List of Tables

3-1	Events and <i>t_look()</i>	15
4-1	Classification of the XTI Functions.....	26
5-1	Transport Interface States.....	28
5-2	Transport Interface Outgoing Events.....	29
5-3	Transport Interface Incoming Events.....	30
5-4	Transport Interface User Actions	31
5-5	Initialisation/De-initialisation States.....	32
5-6	Data Transfer States: Connectionless-mode Service.....	32
5-7	Connection/Release/Data Transfer States: Connection-mode Service....	33
7-1	XTI-level Options	79
A-1	Options for Quality of Service and Expedited Data	190
A-2	Management Options.....	192
A-3	Options for Quality of Service	194
A-4	Management Option.....	195
B-1	TCP-level Options.....	200
B-2	UDP-level Option.....	201

B-3	IP-level Options	202
C-1	Sequence of Transport Functions in Connection-oriented Mode.....	209
C-2	Sequence of Transport Functions in Connectionless Mode.....	210
H-1	APCO-level Options	270
H-2	APCL-level Options.....	271
H-3	Association Establishment	279
H-4	Data Transfer	280
H-5	Association Release.....	281
H-6	Connectionless-mode ACSE Service	282
I-1	SNA Options	297
I-2	Fields for <i>info</i> Parameter	298
I-3	Default Characteristics returned by <i>t_open()</i>	299
I-4	FDX LU 6.2 Verb Definitions.....	302
I-5	XTI Mapping to LU 6.2 Full Duplex Verbs.....	312
I-6	Relation Symbol Description	314
I-7	<i>t_accept</i> <--> FDX Verbs and Parameters	314
I-8	<i>t_bind</i> <--> FDX Verbs and Parameters	315
I-9	<i>t_close</i> <--> FDX Verbs and Parameters	315
I-10	<i>t_connect</i> <--> FDX Verbs and Parameters.....	316
I-11	<i>t_getprocaddr</i> <--> FDX Verbs and Parameters	318
I-12	<i>t_listen</i> <--> FDX Verbs and Parameters	319
I-13	<i>t_optmgmt</i> <--> FDX Verbs and Parameters	319
I-14	<i>t_rcv</i> <--> FDX Verbs and Parameters	320
I-15	<i>t_rcvconnect</i> <--> FDX Verbs and Parameters	321
I-16	<i>t_snd</i> <--> FDX Verbs and Parameters	322
I-17	<i>t_snddis</i> (Existing Connection) <--> FDX Verbs and Parameters	323
I-18	<i>t_snddis</i> (Incoming Connect Req.) <--> FDX Verbs and Parameters.....	323
I-19	<i>t_sndrel</i> <--> FDX Verbs and Parameters	323
I-20	Mapping of XTI Events to SNA Events	325

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to `info-server@xopen.co.uk` with the following in the Subject line:

```
request corrigenda; topic index
```

This will return the index of publications for which Corrigenda exist.

This Document

This document is a CAE Specification (see above). However, an appendix may be a CAE Specification, a Preliminary Specification or a vehicle for conveying information to implementors. In this regard, the introductory section in each appendix clearly identifies its status.

This document centres around three sets of networking interfaces:

1. X/Open Transport Interface (XTI)

Chapter 2 through Chapter 7 define a transport-service interface that allows multiple users to communicate at the transport level of the OSI reference model.

2. Sockets

Chapter 8 and Chapter 9 describe a set of interfaces to process-to-process communication services.

3. IP Address Resolution

Chapter 10 and Chapter 11 describe a set of interfaces that obtain network information and are usable in conjunction with both XTI and Sockets when using the Internet Protocol (IP).

This document incorporates XTI, Version 2 (September 1993)¹.

1. X/Open CAE Specification, September 1993, X/Open Transport Interface (XTI), Version 2 (ISBN: 1-872630-97-9, C318).

Structure

- Chapter 1 contains information comparable to that in the **XSH** specification. It applies to the Sockets and Address Resolution interfaces (see below) if the UNIX compilation environment is in effect.
- Chapter 2 is a general introduction to the X/Open Transport Interface (XTI).
- Chapter 3 provides explanatory notes.
- Chapter 4 gives an overview for XTI.
- Chapter 5 describes the states and events in XTI.
- Chapter 6 describes the use of options.
- Chapter 7 contains reference manual pages for the XTI functions and parameters.
- Chapter 8 gives an overview and interfaces for Sockets.
- Chapter 9 defines the headers for Sockets.
- Chapter 10 gives an overview and interfaces for IP Address Resolution.
- Chapter 11 defines the headers for IP Address Resolution.
- Appendix A describes the protocol-specific information that is relevant for ISO transport providers, including for ISO-over-TCP (RFC 1006).
- Appendix B describes the protocol-specific information that is relevant for TCP and UDP transport providers.
- Appendix C gives guidelines for the use of XTI.
- Appendix D specifies a standard programming interface to NetBIOS transport providers in X/Open-compliant systems, using XTI.
- Appendix E describes how XTI provides refinement of the Transport Level Interface (TLI).
- Appendix F presents a subset of the contents of the `<xti.h>` header file.
- Appendix G lists abbreviations used in this document.
- Appendix H provides a simple API exposing a minimum set of OSI Upper Layers functionality (mOSI).
- Appendix I describes the protocol-specific information and mapping to XTI functions that is relevant for Systems Network Architecture (SNA) transport providers.
- Appendix J contains a brief explanation of the Internet Protocols.

Revision History

The only changes to XTI from XTI, Version 2 (September 1993) are as follows:

- The new Chapter 1 contains information comparable to that in the **XSH** specification. It applies to XTI if the UNIX compilation environment is in effect.
 - The compilation environment is defined.
 - Name space reservations are specified.
- The **SYNOPSIS** sections in the functional interfaces in Chapter 7 have been converted from Common Usage C notation to standard C function prototypes, and function prototypes were added to `<xti.h>`.

XTI (February 1992)² merged the following earlier publications into a single document:

- Revised XTI (December 1990)³
- Addendum to Revised XTI (August 1991)⁴

In XTI, Version 2 (September 1993) the main body was unchanged. It contained additions as follows:

- Chapter 2, Introduction has been extended to explain the role of the appendices in relation to the main body of the XTI specification.
- Appendix A, ISO Transport Protocol Information has been extended to incorporate RFC 1006 (ISO Transport Service on Top of the TCP)
- A new Appendix — Appendix H, Minimum OSI Functionality — has been added.
- A new Appendix — Appendix I, SNA Transport Provider — has been added.
- Appendix I, Glossary of XTI (February 1992) has been retitled simply as Glossary, in accordance with the latest X/Open house style for X/Open specifications.

Similarly, other minor restructuring has been carried out to align with the latest X/Open house style.

The revisions to XTI between publication of the X/Open Portability Guide, Issue 3 (XPG3) and XTI (February 1992) are summarised here in two stages:

1. Those which appeared in Revised XTI (December 1990):

These changes arose principally from implementation experience gathered by X/Open member companies.

Delete optional functions

The concept of mandatory *versus* optional functions is contrary to the goal of portability. Therefore, all XTI functions were made mandatory; [TNOTSUPPORT] should be returned if the transport provider does not support the function requested.

Error messages

The format of messages produced by the `t_error()` function was clarified. See also the additional function `t_strerror()`.

Multiple use of addresses

More stringent recommendations about multiple use of addresses were made. This enhanced portability across different transport providers.

State behaviour

The state machine behaviour of XTI was clarified by the addition of a T_UNBND column in Table 5-7 of Chapter 5, States and Events in XTI, and by the identification of a number of additional cases where asynchronous events resulted in the return of the TLOOK error.

-
2. X/Open CAE Specification, February 1992, X/Open Transport Interface (XTI) (ISBN: 1-872630-29-4, C196 or XO/CAE/91/600).
 3. X/Open Developers' Specification, December 1990, Revised XTI (X/Open Transport Interface) (ISBN: 1-872630-05-7, D060 or XO/DEV/90/060).
 4. X/Open Addendum, August 1991, Addendum to Revised XTI (ISBN: 1-872630-21-9, A110 or XO/AD/91/010).

Zero-length TSDUs and TSDU fragments

The extent of support for zero-length TSDUs and zero-length TSDU fragments was set out more clearly. See the descriptions of functions *t_snd()* and *t_getinfo()* in Chapter 7, XTI Library Functions and Parameters.

T_MORE

The significance of the T_MORE flag for asynchronously received data was clarified. See the description of *t_rcv()* in Chapter 7, XTI Library Functions and Parameters.

Protocol options

The description of protocol options for both OSI and TCP was much enhanced (see Appendix A, ISO Transport Protocol Information and Appendix B, Internet Protocol-specific Information).

Options and management structures

These were extensively revised, especially those covering connection-oriented OSI (see Appendix F, Headers and Definitions).

Expedited Data

The different significance of expedited data in the OSI and TCP cases was clarified.

Connect semantics

Differences in underlying protocol semantics between OSI and TCP at connection establishment were clarified. See Appendix B, Internet Protocol-specific Information and the descriptions of *t_accept()* and *t_listen()* in Chapter 7, XTI Library Functions and Parameters.

Add function *t_getprotaddr()*

This function yields the local and remote protocol addresses currently associated with a transport endpoint.

Add function *t_strerror()*

This function maps an error number into a language-dependent error message string. The functionality corresponds to the error message changes in the *t_error()* function.

Add Valid States to function descriptions

All function descriptions were revised to include an indication of the interface states for which they are valid.

Add new error codes

A number of new error codes were added (see Appendix F, Headers and Definitions).

A number of minor changes were also made, including:

- clarification of the use of the term *socket* in the TCP case
- clarification of support for automatic generation of addresses
- clarification of the management of flow control
- clarification of the significant differences between transport providers
- clarification of the issue of non-guaranteed delivery of data at connection close
- clarification of the ways in which error indications may be received in connectionless working
- enhancement of *t_optmgmt()* to allow retrieval of current value of transport provider options

— addition of *extern* definitions for all XTI functions in Appendix F, Headers and Definitions.

2. Those which appeared in Addendum to Revised XTI (August 1991):

These changes were consolidated into XTI (February 1992). The revisions listed below refer to chapter, section and appendix references in Revised XTI (December 1990).

Section 2.9.1

The *Protocol options* and *Options and management structures* paragraphs were deleted and replaced with the following:

Option management

The management and usage of options were completely revised. The changes affected Chapter 5, the *t_optmgmt()* manual pages in Chapter 6, Appendix A, Appendix B and Appendix F.

Section 4.5

The row for *optmgmt* was deleted from Figure 5, and a new row added to Figure 7 for the event *optmgmt*, as follows:

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

Chapter 5

Chapter 5, Transport Protocol-specific Options was renamed The Use of Options, and previous text replaced with new text.

Chapter 6, *t_accept()*

In the second paragraph, the phrase *protocol-specific parameters* was replaced with *options*.

In the sixth paragraph, the sentence “The values of parameters specified by *opt* and the syntax of those values are protocol-specific.” was removed.

In the seventh paragraph, the phrase *protocol-specific option* was replaced with *option*.

Chapter 6, *t_connect()*

In the sixth paragraph, “If used, *sndcall->opt.buf* must point to the corresponding options structures (**isoco_options** or **tcp_options**);” was replaced with “If used, *sndcall->opt.buf* must point to a buffer with the corresponding options;”.

Chapter 6, *t_listen()*

In the second paragraph, *protocol-specific parameters* was replaced with *options*.

Chapter 6, *t_optmgmt()*

The manual pages for *t_optmgmt()* in Chapter 6 were completely replaced with new text.

Chapter 6, *t_rcvconnect()*

In the third paragraph, *protocol-specific information* was replaced with *options*.

Chapter 6, *t_rcvudata()* and *t_rcvuderr()*

In the third paragraph, *protocol-specific options* was replaced with *options*.

Chapter 6, *t_sndudata()*

In the second paragraph, *protocol-specific options* was replaced with *options*.

Appendix A

The text in Appendix A, ISO Transport Protocol Information was replaced with new text.

Appendix B

The text in Appendix B, Internet Protocol-specific Information was replaced with new text.

Appendix F

The text in Appendix F, Headers and Definitions was replaced with new text.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
 - command operands, command option-arguments or variable names, for example, substitutable argument prototypes
 - environment variables, which are also shown in capitals
 - utility names
 - external variables, such as *errno*
 - functions; these are shown as follows: *name()*. Names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **#define** construct.
- The notation [EABCD] is used to identify a return value ABCD, including if this is an error value.
- Syntax, code examples and user input in interactive examples are shown in fixed width font. Brackets shown in this font, [], are part of the syntax and do *not* indicate optional items. In syntax the | symbol is used to separate alternatives, and ellipses (. . .) are used to show that additional arguments are optional.

Trade Marks

AT&T[®] is a registered trade mark of AT&T in the U.S.A. and other countries.

HP[®] is a registered trade mark of Hewlett-Packard.

OSF[™] is a trade mark of The Open Software Foundation, Inc.

SNA is a product of International Business Machines Corporation.

UNIX[®] is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

/usr/group[®] is a registered trade mark of UniForum, the International Network of UNIX System Users.

X/Open[®] is a registered trade mark, and the “X” device is a trade mark, of X/Open Company Limited.

Acknowledgements

- AT&T for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.
- The Institution of Electrical and Electronics Engineers, Inc. for permission to reproduce portions of its copyrighted IEEE Std 1003.2/D12, which have since become the corresponding portions of IEEE Std 1003.2-1992 and ISO/IEC 9945-2:1993, and also for permission to reproduce portions of IEEE Std P1003.1g/D4.
- The IEEE Computer Society's Portable Applications Standards Committee (PASC), whose Standards contributed to our work.
- The UniForum (formerly /usr/group) Technical Committee's Internationalization Subcommittee for work on internationalised regular expressions.
- The ANSI X3J11 Committees.
- Hewlett-Packard Company, International Business Machines Corporation, Novell Inc., The Open Software Foundation, and Sun Microsystems, Inc., for their work in developing the Single X/Open UNIX Extension and sponsoring it through the X/Open Direct Review (Fast-track) process.

Referenced Documents

The following documents are referenced in this specification:

ACSE

ISO 8649

ISO 8649: 1988 Information Processing Systems — Open Systems Interconnection — Service Definition for the Association Control Service Element, together with:

Technical Corrigendum 1: 1990 to ISO 8649: 1988

Amendment 1: 1990 to ISO 8649: 1988

Authentication during association establishment.

Amendment 2: 1991 to ISO 8649: 1988

Connectionless-mode ACSE Service.

ISO 8650

ISO 8650: 1988 Information Processing Systems — Open Systems Interconnection — Protocol specification for the Association Control Service Element, together with:

Technical Corrigendum 1: 1990 to ISO 8650: 1988

Amendment 1: 1990 to ISO 8650: 1988

Authentication during association establishment.

ISO/IEC 10035

ISO/IEC 10035: 1991, Information Technology — Open Systems Interconnection — Connectionless ACSE Protocol Specification.

Presentation

ISO 8822

ISO 8822: 1988, Information Processing Systems — Open Systems Interconnection — Connection-oriented Presentation Service Definition.

ISO 8823

ISO 8823: 1988, Information Processing Systems — Open Systems Interconnection — Connection-oriented Presentation Protocol Specification.

ISO 8824

ISO 8824: 1990, Information Technology — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1).

BER

ISO/IEC 8825:1990 (ITU-T Recommendation X.209 (1988)), Information Technology — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

ISO/IEC 9576

ISO/IEC 9576: 1991, Information Technology — Open Systems Interconnection — Connectionless Presentation Protocol Specification.

Session

ISO 8326

ISO 8326: 1987, Information Processing Systems — Open Systems Interconnection — Basic Connection-oriented Session Service Definition.

ISO 8327

ISO 8327: 1987, Information Processing Systems — Open Systems Interconnection — Basic Connection-oriented Session Protocol Specification.

Amendment 3: 1992 to ISO 8327: 1987 — Additional Synchronization Functionality.

Other References

Minimal OSI

ISO/IEC DISP 11188-3, International Standardized Profile — Common Upper Layer Requirements — Part 3: Minimal OSI Upper Layers Facilities, 1994-04-14.

ISO 7498

ISO 7498: 1984, Information Processing Systems — Open Systems Interconnection — Basic Reference Model.

ISO Transport

	Connection-Oriented	Connectionless
Protocol Definition	IS 8073-1986	IS 8602
Service Definition	IS 8072-1986	IS 8072/Add.1-1986

TCP

Transmission Control Protocol, RFC 793 (Defense Communication Agency, DDN Protocol Handbook, Volume II, DARPA Internet Protocols, (December 1985). Also see TCP, Transmission Control Protocol, Military Standard, Mil-std-1778, Defense Communication Agency, DDN Protocol Handbook, Volume I, DOD Military Standard Protocols (December 1985).

UDP

User Datagram Protocol, RFC 768 (Defense Communication Agency, DDN Protocol Handbook, Volume II, DARPA Internet Protocols, December 1985).

TLI Specifications

Networking Services Extension, draft version of SVID Issue 2, Volume III, 1986.

NetBIOS

Mappings of NetBIOS services to OSI and IPS transport protocols are provided in the X/Open CAE Specification, October 1992, Protocols for X/Open PC Interworking: SMB, Version 2 (ISBN: 1-872630-45-6, C209).

SNA

SNA National Registry, IBM document G325-6025-0.

CURL

Common Upper Layer Requirements, Part 3: Minimal OSI Upper Layer Facilities — OIW/EWOS working document.

XSH, Issue 4, Version 2

X/Open CAE Specification, August 1994, System Interfaces and Headers, Issue 4, Version 2 (ISBN: 1-85912-037-7, C435).

Referenced Documents

XCU, Issue 4, Version 2

X/Open CAE Specification, August 1994, Commands and Utilities, Issue 4, Version 2 (ISBN: 1-85912-034-2, C436).

Common Information

This chapter provides general information that applies to the XTI, Sockets and IP Address Resolution interfaces defined in this volume.

1.1 Terminology

The information in this section applies only to the Sockets and IP Address Resolution interfaces.

The following terms are used in this specification:

can

This describes a permissible optional feature or behaviour available to the user or application; all systems support such features or behaviour as mandatory requirements.

implementation-dependent

The value or behaviour is not consistent across all implementations. The provider of an implementation normally documents the requirements for correct program construction and correct data in the use of that value or behaviour. When the value or behaviour in the implementation is designed to be variable or customisable on each instantiation of the system, the provider of the implementation normally documents the nature and permissible ranges of this variation. Applications that are intended to be portable must not rely on implementation-dependent values or behaviour.

may

With respect to implementations, the feature or behaviour is optional. Applications should not rely on the existence of the feature. To avoid ambiguity, the reverse sense of *may* is expressed as *need not*, instead of *may not*.

must

This describes a requirement on the application or user.

obsolescent

Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this document. They are retained in this version because of their widespread use. Their use in new applications is discouraged.

should

With respect to implementations, the feature is recommended, but it is not mandatory. Applications should not rely on the existence of the feature.

With respect to users or applications, the word means recommended programming practice that is necessary for maximum portability.

undefined

A value or behaviour is undefined if this document imposes no portability requirements on applications for erroneous program constructs or erroneous data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application using such behaviour is not fully portable to all systems.

unspecified

A value or behaviour is unspecified if this document imposes no portability requirements on applications for correct program construct or correct data. Implementations may specify the

42 result of using that value or causing that behaviour, but such specifications are not guaranteed
43 to be consistent across all implementations. An application requiring a specific behaviour,
44 rather than tolerating any behaviour when using that functionality, is not fully portable to all
45 systems.

46 **will**

47 This means that the behaviour described is a requirement on the implementation and
48 applications can rely on its existence.

49 1.1.1 Shaded Text

50 Shaded text in this document is qualified by a code in the left margin. The code and its meaning
51 is as follows:

52 UX **X/Open UNIX Extension**

53 The material relates to interfaces included to provide portability for applications originally
54 written to be compiled on UNIX and UNIX-based operating systems. Therefore, the features
55 described may not be present on systems that conform to XPG4 or to earlier XPG releases. The
56 relevant reference manual pages may provide additional or more specific portability warnings
57 about use of the material.

58 If an entire **SYNOPSIS** section is shaded and marked with one UX, all the functionality described
59 in that entry is an extension.

60 The material on pages labelled X/OPEN UNIX and the material flagged with the UX margin
61 legend is available only in cases where the `_XOPEN_UNIX` version test macro is defined.

62 1.2 Use and Implementation of Interfaces

63 UX The requirements in the remainder of this chapter are in effect only if the application has defined
64 `XOPEN_SOURCE_EXTENDED = 1`.

65 Each of the following statements applies unless explicitly stated otherwise in the detailed
66 descriptions that follow. If an argument to a function has an invalid value (such as a value
67 outside the domain of the function, or a pointer outside the address space of the program, or a
68 null pointer), the behaviour is undefined. Any function declared in a header may also be
69 implemented as a macro defined in the header, so a library function should not be declared
70 explicitly if its header is included. Any macro definition of a function can be suppressed locally
71 by enclosing the name of the function in parentheses, because the name is then not followed by
72 the left parenthesis that indicates expansion of a macro function name. For the same syntactic
73 reason, it is permitted to take the address of a library function even if it is also defined as a
74 macro. The use of the C-language `#undef` construct to remove any such macro definition will
75 also ensure that an actual function is referred to. Any invocation of a library function that is
76 implemented as a macro will expand to code that evaluates each of its arguments exactly once,
77 fully protected by parentheses where necessary, so it is generally safe to use arbitrary
78 expressions as arguments. Likewise, those function-like macros described in the following
79 sections may be invoked in an expression anywhere a function with a compatible return type
80 could be called.

81 Provided that a library function can be declared without reference to any type defined in a
82 header, it is also permissible to declare the function, either explicitly or implicitly, and use it
83 without including its associated header. If a function that accepts a variable number of
84 arguments is not declared (explicitly or by including its associated header), the behaviour is
85 undefined.

86 As a result of changes in this issue of this document, application writers are only required to
87 include the minimum number of headers. Implementations of XSI-conformant systems will
88 make all necessary symbols visible as described in the Headers section of this document.

89 **1.2.1 C Language Definition**

90 The C language that is the basis for the synopses and code examples in this document is *ISO C*,
91 as specified in the referenced ISO C standard. *Common Usage C*, which refers to the C language
92 before standardisation, was the basis for previous editions of the **XTI** specification.

93 1.3 The Compilation Environment

94 Applications should ensure that the feature test macro `_XOPEN_SOURCE` is defined before
95 inclusion of any header. This is needed to enable the functionality described in this document
96 (but see also Section 1.3.1), and possibly to enable functionality defined elsewhere in the
97 Common Applications Environment.

98 The `_XOPEN_SOURCE` macro may be defined automatically by the compilation process, but to
99 ensure maximum portability, applications should make sure that `_XOPEN_SOURCE` is defined
100 by using either compiler options or `#define` directives in the source files, before any `#include`
101 directives. Identifiers in this document may be undefined using the `#undef` directive as
102 described in Section 1.2 on page 2 or Section 1.3.2 on page 5. These `#undef` directives must
103 follow all `#include` directives of any XSI headers.

104 Most strictly conforming POSIX and ISO C applications will compile on systems compliant to
105 this specification. However, an application which uses any of the items marked as an extension
106 to POSIX and ISO C, for any purpose other than that shown here, may not compile. In such
107 cases, it may be necessary to alter those applications to use alternative identifiers.

108 Since this document is aligned with the ISO C standard, and since all functionality enabled by
109 having `_POSIX_C_SOURCE` set equal to 2 should be enabled by `_XOPEN_SOURCE`, there
110 should be no need to define either `_POSIX_SOURCE` or `_POSIX_C_SOURCE` if
111 `_XOPEN_SOURCE` is defined. Therefore, if `_XOPEN_SOURCE` is defined and
112 `_POSIX_SOURCE` is defined, or `_POSIX_C_SOURCE` is set equal to 1 or 2, the behaviour is the
113 same as if only `_XOPEN_SOURCE` is defined. However, should `_POSIX_C_SOURCE` be set to a
114 value greater than 2, the behaviour is undefined.

115 The *c89* and *cc* utilities recognise the additional `-I` operand for standard libraries:

116 `-I xnet` If the implementation defines `_XOPEN_UNIX`, this operand makes visible all
117 functions referenced in this document. An implementation may search this library
118 in the absence of this operand.

119 It is unspecified whether the library `libxnet.a` exists as a regular file.

120 If the implementation supports the utilities marked **DEVELOPMENT** in the XCU specification,
121 the *lint* utility recognises the additional `-I` operand for standard libraries:

122 `-I xnet` Names the library `llib-lxnet.ln`, which will contain functions specified in this
123 document.

124 It is unspecified whether the library `llib-lxnet.ln` exists as a regular file.

125 1.3.1 X/Open UNIX Extension

126 An application that relies on any portion of this specification must define
127 `_XOPEN_SOURCE_EXTENDED = 1` in each source file or as part of its compilation
128 environment. When `_XOPEN_SOURCE_EXTENDED = 1` is defined in a source file in addition
129 to `_XOPEN_SOURCE`, it must appear before any header is included. The compilation
130 environment will not automatically define the `_XOPEN_SOURCE_EXTENDED` macro.

131 1.3.2 The X/Open Name Space

132 All identifiers in this document are defined in at least one of the headers, as shown in Chapter 9,
 133 Chapter 11 and Appendix F. When `_XOPEN_SOURCE` is defined, each header defines or
 134 declares some identifiers, potentially conflicting with identifiers used by the application. The set
 135 of identifiers visible to the application consists of precisely those identifiers from the header
 136 pages of the included headers, as well as additional identifiers reserved for the implementation.
 137 In addition, some headers may make visible identifiers from other headers as indicated on the
 138 relevant header pages.

139 The identifiers reserved for use by the implementation are described below.

- 140 1. Each identifier with external linkage described in the header section is reserved for use as
 141 an identifier with external linkage if the header is included.
- 142 2. Each macro name described in the header section is reserved for any use if the header is
 143 included.
- 144 3. Each identifier with file scope described in the header section is reserved for use as an
 145 identifier with file scope in the same name space if the header is included.

146 If any header in the following table is included, identifiers with the following prefixes or suffixes
 147 shown are reserved for any use by the implementation.

148

149

Header	Prefix	Suffix
<arpa/inet.h>	in_, inet_	
<errno.h>	E	
<netdb.h>	h_, n_, p_, s_	
<netinet/in.h>	in_, s_, sin_	
<sys/socket.h>	sa_, if_, ifc_, ifru_, infu_, ifra_, msg_, cmsg_, l_	
<sys/un.h>	sun_	
<xti.h>	l_, t_, T	
ANY header		_t

150

151

152

153

154

155

156

157

158 If any header in the following table is included, macros with the prefixes shown may be defined.
 159 After the last inclusion of a given header, an application may use identifiers with the
 160 corresponding prefixes for its own purpose, provided their use is preceded by an `#undef` of the
 161 corresponding macro.

162

163

Header	Prefix
<netinet/in.h>	IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_
<sys/socket.h>	AF_, MSG_, PF_, SO
<xti.h>	INET_, IP_, ISO_, OPT_, T_, TCL_, TCP_, TCO_, XTI_

164

165

166

167 The following identifiers are reserved regardless of the inclusion of headers:

- 168 1. All identifiers that begin with an underscore and either an upper-case letter or another
 169 underscore are always reserved for any use by the implementation.
- 170 2. All identifiers that begin with an underscore are always reserved for use as identifiers with
 171 file scope in both the ordinary identifier and tag name spaces.

172 3. All identifiers in the table below are reserved for use as identifiers with external linkage.

173
174
175
176
177
178
179
180

XTI:				
t_accept	t_error	t_look	t_rcvrel	t_sndudata
t_alloc	t_free	t_open	t_rcvudata	t_strerror
t_bind	t_getinfo	t_optmgmt	t_rcvuderr	t_sync
t_close	t_getprotaddr	t_rcv	t_snd	t_unbind
t_connect	t_getstate	t_rcvconnect	t_snddis	
t_errno	t_listen	t_rcvdis	t_sndrel	
Sockets:				
accept	getsockname	recvfrom	sendto	socketpair
bind	getsockopt	recvmsg	setsockopt	
connect	listen	send	shutdown	
getpeername	recv	sendmsg	socket	
IP Address Resolution:				
endhostent	inet_lnaof	endservent	h_errno	gethostent
getprotoent	sethostent	getnetbyname	inet_network	getprotobynumber
inet_addr	endprotoent	getservent	setservent	htons
ntohs	getnetbyaddr	inet_netof	gethostbyname	ntohl
endnetent	getservbyport	setprotoent	getprotobyname	
gethostname	inet_makeaddr	gethostbyaddr	htonl	
getservbyname	setnetent	getnetent	inet_ntoa	

181
182
183
184
185
186
187
188
189
190
191
192
193

194 All the identifiers defined in this document that have external linkage are always reserved for
195 use as identifiers with external linkage.

196 No other identifiers are reserved.

197 Applications must not declare or define identifiers with the same name as an identifier reserved
198 in the same context. Since macro names are replaced whenever found, independent of scope and
199 name space, macro names matching any of the reserved identifier names must not be defined if
200 any associated header is included.

201 Headers may be included in any order, and each may be included more than once in a given
202 scope, with no difference in effect from that of being included only once.

203 If used, a header must be included outside of any external declaration or definition, and it must
204 be first included before the first reference to any type or macro it defines, or to any function or
205 object it declares. However, if an identifier is declared or defined in more than one header, the
206 second and subsequent associated headers may be included after the initial reference to the
207 identifier. Prior to the inclusion of a header, the program must not define any macros with
208 names lexically identical to symbols defined by that header.

209 **1.4 Relationship to the XSH Specification**

210 **1.4.1 Error Numbers**

211 Some functions provide an error number in *errno*, which is either a variable or macro defined in
212 `<errno.h>`; the macro expands to a modifiable lvalue of type `int`.

213 A list of valid values for *errno* and advice to application writers on the use of *errno* appears in the
214 **XSH** specification.

215 **1.5 Relationship to Emerging Formal Standards**

216 The IEEE 1003.8 standards committee is also developing interfaces to XTI and Sockets. X/Open
217 is actively involved in the work of this committee.

General Introduction to the XTI

218

219 The X/Open Transport Interface (XTI) specification defines an independent transport-service
 220 interface that allows multiple users to communicate at the transport level of the OSI reference
 221 model. The specification describes transport-layer characteristics that are supported by a wide
 222 variety of transport-layer protocols. Supported characteristics include:

- 223 • connection establishment
- 224 • state change support
- 225 • event handling
- 226 • data transfer
- 227 • option manipulation.

228 Although all transport-layer protocols support these characteristics, they vary in their level of
 229 support and/or their interpretation and format. For example, there are transport-level options
 230 which remain constant across all transport providers while there are other options which are
 231 transport-provider specific or have different values/names for different transport providers.

232 The main Chapters in this specification describe interfaces, parameters and semantics constant
 233 across all transport providers. The remainder of the document consists of appendices that
 234 provide valuable information that is not an integral part of the main body since it is either
 235 descriptive or applies only to some transport providers.

236 Some appendices provide information pertinent to writing XTI applications over specific
 237 transport providers. The transport providers fall into three classes:

- 238 • Those corresponding to traditional transport providers, such as:
 - 239 — ISO Transport (connection-oriented or connectionless)
 - 240 — TCP
 - 241 — UDP
 - 242 — NetBIOS.
- 243 • Those corresponding to commonly used subsets of higher-layer protocols that provide
 244 transport-like services, such as:
 - 245 — minimal functionality OSI (mOSI), that is, OSI ACSE/Presentation with the kernel and
 246 duplex functional units
 - 247 — SNA LU6.2 subset.
- 248 • Mixed-protocol providers that provide the appearance of one protocol over a different
 249 protocol such as:
 - 250 — ISO transport appearance (connection-oriented) over TCP.

251 The ISO appendix (Appendix A) also describes a transport provider that uses RFC 1006 to
 252 compensate for the differences between ISO transport and TCP so that a TCP provider
 253 can present an ISO transport appearance.

254 While XTI gives transport users considerable independence from the underlying transport
255 provider, the differences between providers are not entirely hidden. Appendix C includes
256 guidelines for writing transport-provider-independent software, which can be done primarily by
257 using only functions supported by all providers, avoiding option management, and using a
258 provider-independent means of acquiring addresses.

259 While the transport-provider-specific appendices are intended mostly for transport users, they
260 are also used by implementors of transport providers. For the purposes of the implementors,
261 some of the appendices show how XTI services can be mapped to primitives associated with the
262 specific providers. These are provided as guidance only and do not dictate anything about a
263 given implementation.

264 Some of the appendices to the XTI specification are included as vehicles for communicating
265 information needed by implementors, or guidelines to the use of the specification in question.
266 The Guidelines for the use of XTI (Appendix C), Minimum OSI Functionality (Appendix H),
267 SNA Transport provider (Appendix I) and comparison of XTI to TLI (Appendix E) belong to this
268 category.

269 Some other appendices, however, have evolved into a prescriptive specification, as in the case of
270 Appendix A for the ISO transport provider, Appendix B for the Internet transport provider and
271 Appendix D for the NetBIOS transport provider. Since not every XTI implementor would find it
272 relevant to implement the functionality of all of these appendices, they have been kept separate
273 from the main XTI specification, thus becoming brandable XTI options. Support for these
274 transport providers is declared by vendors through the XTI Conformance Statement
275 Questionnaire.

276 An appendix may have a different status from the overall XTI specification. Thus the appendix
277 for a particular transport provider may be a Preliminary Specification while the document is a
278 CAE specification. When this is the case, the status of the appendix is clearly identified in its
279 own introduction.

280 Topics beyond the scope of the XTI specification include:

- 281 • Address parameters

282 Several functions have parameters for addresses. The structure of these addresses is beyond
283 the scope of this document. Specific implementations specify means for transport users to get
284 or construct addresses.

- 285 • Event management

286 In order for applications to use XTI in a fully asynchronous manner, it will be necessary for
287 the application to include facilities of an Event Management (EM) interface. Such EM facility
288 may allow the application to be notified of a number of events over a range of active
289 transport connections. For example, one event may denote a connection is flow-controlled.
290 While Appendix C provides some guidelines for using EM in XTI applications, a complete
291 specification defining an EM interface is beyond the scope of this document.

3.1 Transport Endpoints

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

302 To be active, a transport endpoint must have a transport address associated with it by the
303 *t_bind()* function. A transport connection is characterised by the association of two active
304 endpoints, made by using the functions of establishment of transport connection. The *fd* is a
305 communication path to a transport provider. There is no direct assignation of the processes to
306 the transport provider, so multiple processes, which obtain the *fd* by *open()*, *fork()* or *dup()*
307 operations, may access a given communication path. Note that the *open()* function will work
308 only if the opened character string is a pathname.

309 Note that in order to guarantee portability, the only operations which the applications may
310 perform on any *fd* returned by *t_open()* are those defined by XTI and *fcntl()*, *dup()* or *dup2()*.
311 Other operations are permitted but these will have system-dependent results.

3.2 Transport Providers

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

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

321 **3.3 Association of a UNIX Process to an Endpoint**

322 One process can simultaneously open several *fds*. However, in synchronous mode, the process
323 must manage the different actions of the associated transport connections sequentially.
324 Conversely, several processes can share the same *fd* (by *fork()* or *dup()* operations) but they have
325 to synchronise themselves so as not to issue a function that is unsuitable to the current state of
326 the transport endpoint.

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

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

339 **3.4 Use of the Same Protocol Address**

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

343 3.5 Modes of Service

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

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

354 In contrast, the connectionless-mode transport service is message-oriented and supports data
355 transfer in self-contained units with no logical relationship required among multiple units.
356 These units are also known as datagrams. This service requires a pre-existing association
357 between the peer users involved, which determines the characteristics of the data to be
358 transmitted. No dynamic negotiation of parameters and options is supported by this service.
359 All the information required to deliver a unit of data (for example, destination address) is
360 presented to the transport provider, together with the data to be transmitted, in a single service
361 access which need not relate to any other service access. Also, each unit of data transmitted is
362 entirely self-contained, and can be independently routed by the transport provider. This service
363 is attractive to applications that involve short-term request/response interactions, exhibit a high
364 level of redundancy, are dynamically reconfigurable or do not require guaranteed, in-sequence
365 delivery of data.

366 3.6 Error Handling

367 Two levels of error are defined for the transport interface. The first is the library error level.
368 Each library function has one or more error returns. Failures are indicated by a return value of
369 -1 . An external integer, *t_errno*, which is defined in the header `<xti.h>`, holds the specific error
370 number when such a failure occurs. This value is set when errors occur but is not cleared on
371 successful library calls, so it should be tested only after an error has been indicated. If
372 implemented, a diagnostic function, *t_error()*, prints out information on the current transport
373 error. The state of the transport provider may change if a transport error occurs.

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

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

383 3.7 Synchronous and Asynchronous Execution Modes

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

388 The transport service interface supports two execution modes for handling asynchronous
389 events: synchronous mode and asynchronous mode. In the synchronous mode of operation, the
390 transport primitives wait for specific events before returning control to the user. While waiting,
391 the user cannot perform other tasks. For example, a function that attempts to receive data in
392 synchronous mode will wait until data arrives before returning control to the user. Synchronous
393 mode is the default mode of execution. It is useful for user processes that want to wait for
394 events to occur, or for user processes that maintain only a single transport connection. Note that
395 if a signal arrives, blocking calls are interrupted and return a negative return code with *t_errno*
396 set to [TSYSERR] and *errno* set to [EINTR]. In this case the call will have no effect.

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

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

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

- 415 • T_LISTEN
- 416 • T_CONNECT
- 417 • T_DATA
- 418 • T_EXDATA
- 419 • T_DISCONNECT
- 420 • T_ORDREL
- 421 • T_UDERR
- 422 • T_GODATA
- 423 • T_GOEXDATA.

424 These are described in Section 3.8 on page 16.

425 A process that issues functions in synchronous mode must still be able to recognise certain
426 asynchronous events and act on them if necessary. This is handled through a special transport

427 error [TLOOK] which is returned by a function when an asynchronous event occurs. The
 428 *t_look()* function is then invoked to identify the specific event that has occurred when this error
 429 is returned.

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

434 **Events and *t_look()***

435 All events that occur at a transport endpoint are stored by XTI. These events are retrievable one
 436 at the time via the *t_look()* function. If multiple events occur, it is implementation-dependent in
 437 what order *t_look()* will return the events. An event is outstanding on a transport endpoint until
 438 it is consumed. Every event has a corresponding consuming function which handles the event and
 439 clears it. Both T_DATA and T_EXDATA events are consumed when the corresponding
 440 consuming function has read all the corresponding data associated with that event. The
 441 intention of this is that T_DATA should always indicate that there is data to receive. Two events,
 442 T_GODATA and T_GOEXDATA, are also cleared as they are returned by *t_look()*. Table 3-1
 443 summarises this.

444 Event	445 Cleared on <i>t_look()</i> ?	446 Consuming XTI functions
447 T_LISTEN	448 No	449 <i>t_listen()</i>
450 T_CONNECT	451 No	452 <i>t_{rcv}connect()</i> *
453 T_DATA	454 No	455 <i>t_rcv{udata}()</i>
456 T_EXDATA	457 No	458 <i>t_rcv()</i>
459 T_DISCONNECT	460 No	461 <i>t_rcvdis()</i>
462 T_UDERR	463 No	464 <i>t_rcvuderr()</i>
465 T_ORDREL	466 No	467 <i>t_rcvrel()</i>
468 T_GODATA	469 Yes	470 <i>t_snd{udata}()</i>
471 T_GOEXDATA	472 Yes	473 <i>t_snd()</i>

455 **Table 3-1** Events and *t_look()*

456 _____
 457 * In the case of the *t_connect()* function the T_CONNECT event is both generated and consumed by the execution of the function
 458 and is therefore not visible to the application.

459 3.8 Event Management

460 Each XTI call deals with one transport endpoint at a time. It is not possible to wait for several
 461 events from different sources, particularly from several transport connections at a time. We
 462 recognise the need for this functionality which may be available today in a system-dependent
 463 fashion.

464 Throughout the document we refer to an event management service called Event Management
 465 (EM) which provides those functions useful to XTI. This Event Management will allow a
 466 process to be notified of the following events:

467	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:
468		1. The file descriptor is bound to a valid address.
469		2. No transport connection is established at this time.
470		
471		
472	T_CONNECT	In connection mode only; a connect response was received by the transport provider; occurs after a <i>t_connect()</i> has been issued.
473		
474	T_DATA	Normal data (whole or part of Transport Service Data Unit (TSDU)) was received by the transport provider.
475		
476	T_EXDATA	Expedited data was received by the transport provider.
477	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 connection establishment functions and on the <i>t_snddis()</i> function.
478		
479		
480	T_ORDREL	An orderly release request was received by a transport provider (connection mode with orderly release only).
481		
482	T_UDERR	In connectionless mode only; an error was found in a previously sent datagram. It may be notified on the <i>t_rcvudata()</i> or <i>t_unbind()</i> function calls.
483		
484		
485	T_GODATA	Flow control restrictions on normal data flow that led to a [TFLOW] error have been lifted. Normal data may be sent again.
486		
487	T_GOEXDATA	Flow control restrictions on expedited data flow that led to a [TFLOW] error have been lifted. Expedited data may be sent again.
488		

490 4.1 Overview of Connection-oriented Mode

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

- 492 • Initialisation/De-initialisation
- 493 • Connection Establishment
- 494 • Data Transfer
- 495 • Connection Release.

496 A state machine is described in Section C.1 on page 207, and the figure in Section C.2 on page
497 208, which defines the legal sequence in which functions from each phase may be issued.

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

- 499 1. supply a *transport provider identifier* for the appropriate type of transport provider (using
500 *t_open()*); this establishes a transport endpoint through which the user may communicate
501 with the provider
- 502 2. associate (bind) an address with this endpoint (using *t_bind()*)
- 503 3. use the appropriate connection functions (using *t_connect()*, or *t_listen()* and *t_accept()*) to
504 establish a transport connection; the set of functions depends on whether the user is an
505 initiator or responder
- 506 4. once the connection is established, normal, and if authorised, expedited data can be
507 exchanged; of course, expedited data may be exchanged only if:
 - 508 • the provider supports it
 - 509 • its use is not precluded by the selection of protocol characteristics; for example, the use
510 of Class 0
 - 511 • negotiation as to its use has been agreed between the two peer transport providers.

512 The semantics of expedited data may be quite different for different transport providers.
513 XTI's notion of expedited data has been defined as the lowest reasonable common
514 denominator.

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

519 **4.1.1 Initialisation/De-initialisation Phase**

520 The functions that support initialisation/de-initialisation tasks are described below. All such
 521 functions provide local management functions; no information is sent over the network.

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

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

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

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

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

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

537 *t_getprotaddr()* This function returns the addresses (local and remote) associated with the
 538 specified transport endpoint.

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

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

542 *t_sync()* This function synchronises the data structures managed by the transport
 543 library with the transport provider.

544 *t_alloc()* This function allocates storage for the specified library data structure.

545 *t_free()* This function frees storage for a library data structure that was allocated by
 546 *t_alloc()*.

547 *t_error()* This function prints out a message describing the last error encountered
 548 during a call to a transport library function.

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

551 *t_strerror()* This function maps an XTI error into a language-dependent error message
 552 string.

553 4.1.2 Overview of Connection Establishment

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

557 In connection mode:

- 558 • The user has first to establish an endpoint; that is, to open a communications path between
 559 the application and the transport provider.
- 560 • Once established, an endpoint must be bound to an address and more than one endpoint
 561 may be bound to the same address. A transport user can determine the addresses associated
 562 with a connection using the `t_getprotaddr()` function.
- 563 • An endpoint can be associated with one, and only one, established transport connection.
- 564 • It is possible to use an endpoint to receive and enqueue incoming connect indications (only if
 565 the provider is able to accept more than one outstanding connect indication; this mode of
 566 operation is declared at the time of calling `t_bind()` by setting `qlen` greater than 0). However,
 567 if more than one endpoint is bound to the same address, only one of them may be used in
 568 this way.
- 569 • The `t_listen()` function is used to look for an enqueued connect indication; if it finds one (at
 570 the head of the queue), it returns details of the connect indication, and a local sequence
 571 number which uniquely identifies this indication, or it may return a negative value with
 572 `t_errno` set to [TNODATA]. The number of outstanding connect requests to dequeue is
 573 limited by the value of the `qlen` parameter accepted by the transport provider on the `t_bind()`
 574 call.
- 575 • If the endpoint has more than one connect indication enqueued, the user should dequeue all
 576 connect indications (and disconnect indications) before accepting or rejecting any or all of
 577 them. The number of outstanding connect indications is limited by the value of the `qlen`
 578 parameter accepted by the transport provider on the call to `t_bind()`.
- 579 • When accepting a connect indication, the transport service user may issue the accept on the
 580 same (listening) endpoint or on a different endpoint.

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

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

- 588 • If the user issues a `t_connect()` on a listening endpoint, again, that endpoint can no longer be
 589 used to receive and enqueue incoming connect requests.
- 590 • A connect attempt failure will result in a value -1 returned from either the `t_connect()` or
 591 `t_rcvconnect()` call, with `t_errno` set to [TLOOK] indicating that a [T_DISCONNECT] event
 592 has arrived. In this case, the reason for the failure may be identified by issuing a `t_rcvdis()`
 593 call.

594 The functions that support these operations of connection establishment are:

595 `t_connect()` This function requests a connection to the transport user at a specified
 596 destination and waits for the remote user's response. This function may be
 597 executed in either synchronous or asynchronous mode. In synchronous

598		mode, the function waits for the remote user's response before returning
599		control to the local user. In asynchronous mode, the function initiates
600		connection establishment but returns control to the local user before a
601		response arrives.
602	<code>t_rcvconnect()</code>	This function enables an active transport user to determine the status of a
603		previously sent connect request. If the request was accepted, the connection
604		establishment phase will be complete on return from this function. This
605		function is used in conjunction with <code>t_connect()</code> to establish a connection in an
606		asynchronous manner.
607	<code>t_listen()</code>	This function enables the passive transport user to receive connect indications
608		from other transport users.
609	<code>t_accept()</code>	This function is issued by the passive user to accept a particular connect
610		request after an indication has been received.

611 4.1.3 Overview of Data Transfer

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

615	<code>t_snd()</code>	This function enables transport users to send either normal or expedited data
616		over a transport connection.
617	<code>t_rcv()</code>	This function enables transport users to receive either normal or expedited
618		data on a transport connection.

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

622 Receiving Data

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

- 627 • Asynchronous Mode

628 The call returns immediately, indicating failure. The user must continue to “poll” for
 629 incoming data, either by issuing repeated call to `t_rcv()`, or by using the `t_look()` or the EM
 630 interface.

- 631 • Synchronous Mode

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

- 633 — Data (normal or expedited) is received.
- 634 — A disconnect indication is received.
- 635 — A signal has arrived.

636 The user may issue a `t_look()` or use EM calls, to determine if data is available.

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

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

647 **Sending Data**

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

- 654 • **Asynchronous Mode**

655 The call returns immediately indicating failure. If the failure was due to flow control
 656 restrictions, then it is possible that only part of the data will actually be accepted by the
 657 transport provider. In this case `t_snd()` will return a value that is less than the number of
 658 octets requested to be sent. The user may either retry the call to `t_snd()` or first receive
 659 notification of the clearance of the flow control restriction via either `t_look()` or the EM
 660 interface, then retry the call. The user may retry the call with the data remaining from the
 661 original call or with more (or less) data, and with the T_MORE flag set appropriately to
 662 indicate whether this is now the end of the TSDU.

- 663 • **Synchronous Mode**

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

- 665 — The flow control restrictions are cleared and the transport provider is able to accept a new
 666 data unit. The `t_snd()` function then returns successfully.
- 667 — A disconnect indication is received. In this case the `t_snd()` function returns
 668 unsuccessfully with `t_errno` set to [TLOOK]. The user can issue a `t_look()` function to
 669 determine the cause of the error. For this particular case `t_look()` will return a
 670 T_DISCONNECT event. Data that was being sent will be lost.
- 671 — An internal problem occurs. In this case the `t_snd()` function returns unsuccessfully with
 672 `t_errno` set to [TSYSERR]. Data that was being sent will be lost.

673 For some transport providers, normal data and expedited data constitute two distinct flows of
 674 data. If either flow is blocked, the user may nevertheless continue using the other one, but in
 675 synchronous mode a second process is needed. The user may send expedited data between the
 676 fragments of a normal TSDU, that is, a `t_snd()` call with the T_EXPEDITED flag set may follow a
 677 `t_snd()` with the T_MORE flag set and the T_EXPEDITED flag not set.

678 Note that XTI supports two modes of sending data, record-oriented and stream-oriented. In the
 679 record-oriented mode, the concept of TSDU is supported, that is, message boundaries are
 680 preserved. In stream-oriented mode, message boundaries are not preserved and the concept of a
 681 TSDU is not supported. A transport user can determine the mode by using the `t_getinfo()`
 682 function, and examining the `tsdu` field. If `tsdu` is greater than zero, this indicates that record-
 683 oriented mode is supported and the return value indicates the maximum TSDU size. If `tsdu`
 684 is zero, this indicates that stream-oriented transfer is supported. For more details see `t_getinfo()` on
 685 page 63.

686 **4.1.4 Overview of Connection Release**

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

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

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

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

708 The functions that support connection release are:

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

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

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

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

721 4.2 Overview of Connectionless Mode

722 The connectionless-mode transport service consists of two phases of communication:
 723 initialisation/de-initialisation and data transfer. A brief description of each phase and its
 724 associated functions is presented below. A state machine is described in Section C.1 on page
 725 207, and the figure in Section C.3 on page 210, that defines the legal sequence in which functions
 726 from each phase may be issued.

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

- 728 1. supply a transport endpoint for the appropriate type of provider (using `t_open()`); this
 729 establishes a transport endpoint through which the user may communicate with the
 730 provider
- 731 2. associate (bind) an address with this transport endpoint (using `t_bind()`)
- 732 3. the user may then send and/or receive connectionless data, as required, using the
 733 functions `t_sndudata()` and `t_rcvudata()`. Once the data transfer phase is finished, the
 734 application may either directly close the file descriptor returned by `t_open()` (using
 735 `t_close()`), thereby freeing the resource for future use, or start a new exchange of data after
 736 disassociating the old address and binding a new one.

737 4.2.1 Initialisation/De-initialisation Phase

738 The functions that support the initialisation/de-initialisation tasks are the same functions used
 739 in the connection-mode service.

740 4.2.2 Overview of Data Transfer

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

- | | | |
|-----|---------------------------|---|
| 743 | <code>t_sndudata()</code> | This function enables transport users to send a self-contained data unit to the user at the specified protocol address. |
| 744 | | |
| 745 | <code>t_rcvudata()</code> | This function enables transport users to receive data units from other users. |
| 746 | <code>t_rcvuderr()</code> | This function enables transport users to retrieve error information associated with a previously sent data unit. |
| 747 | | |

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

750 Receiving Data

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

- 754 • Asynchronous Mode

755 The call returns immediately indicating failure. The user must either retry the call
 756 repeatedly, or “poll” for incoming data by using the EM interface or the `t_look()` function so
 757 as not to be blocked.

- 758 • Synchronous Mode

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

760 — A datagram is received.

761 — An error is detected by the transport provider.

762 — A signal has arrived.

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

765 **Sending Data**

766 • Synchronous Mode

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

770 • Asynchronous Mode

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

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

777 4.3 XTI Features

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

780 `t_bind()`
781 `t_close()`
782 `t_look()`
783 `t_open()`
784 `t_sync()`
785 `t_unbind()`

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

788 `t_accept()`
789 `t_connect()`
790 `t_listen()`
791 `t_rcv()`
792 `t_rcvconnect()`
793 `t_rcvdis()`
794 `t_snd()`
795 `t_snddis()`

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

798 `t_rcvudata()`
799 `t_rcvuderr()`
800 `t_sndudata()`

801 Mandatory mechanisms:

- 802 • synchronous mode
- 803 • asynchronous mode.

804 Utility functions:

805 `t_alloc()`
806 `t_free()`
807 `t_error()`
808 `t_getprotaddr()`
809 `t_getinfo()`
810 `t_getstate()`
811 `t_optmgmt()`
812 `t_strerror()`

813 The orderly release mechanism (using `t_sndrel()` and `t_rcvrel()`), is supported only for
814 T_COTS_ORD type providers. Use with other providers will cause the [TNOTSUPPORT] error
815 to be returned. The use of orderly release is definitely not recommended in order to make
816 applications using TCP portable onto the ISO Transport Layer.

817 Optional mechanisms:

- 818 • the ability to manage (enqueue) more than one incoming connect indication at any one time
- 819 • the address of the caller passed with `t_accept()` may optionally be checked by an XTI
820 implementation.

821 **4.3.1 XTI Functions versus Protocols**

822 Table 4-1 presents all the functions defined in XTI. The character “x” indicates that the mapping
 823 of that function is possible onto a Connection-oriented or Connectionless Transport Service. The
 824 table indicates the type of utility functions as well.

825
826
827
828

Functions	Necessary for Protocol		Utility Functions	
	Connection Oriented	Connectionless	General	Memory
<i>t_accept()</i>	x			
<i>t_alloc()</i>				x
<i>t_bind()</i>	x	x		
<i>t_close()</i>	x	x		
<i>t_connect()</i>	x			
<i>t_error()</i>			x	
<i>t_free()</i>				x
<i>t_getprotaddr()</i>			x	
<i>t_getinfo()</i>			x	
<i>t_getstate()</i>			x	
<i>t_listen()</i>	x			
<i>t_look()</i>	x	x		
<i>t_open()</i>	x	x		
<i>t_optmgmt()</i>			x	
<i>t_rcv()</i>	x			
<i>t_rcvconnect()</i>	x			
<i>t_rcvdis()</i>	x			
<i>t_rcvrel()</i>	x			
<i>t_rcvudata()</i>		x		
<i>t_rcvuderr()</i>		x		
<i>t_snd()</i>	x			
<i>t_snddis()</i>	x			
<i>t_sndrel()</i>	x			
<i>t_sndudata()</i>		x		
<i>t_strerror()</i>			x	
<i>t_sync()</i>			x	
<i>t_unbind()</i>	x	x		

856

Table 4-1 Classification of the XTI Functions

States and Events in XTI

857

858 Table 5-1 through Table 5-7 are included to describe the possible states of the transport provider
859 as seen by the transport user, to describe the incoming and outgoing events that may occur on
860 any connection, and to identify the allowable sequence of function calls. Given a current state
861 and event, the transition to the next state is shown as well as any actions that must be taken by
862 the transport user.

863 The allowable sequence of functions is described in Table 5-5, Table 5-6 and Table 5-7. The
864 support functions, *t_getprotaddr()*, *t_getstate()*, *t_getinfo()*, *t_alloc()*, *t_free()*, *t_look()* and
865 *t_sync()*, are excluded from the state tables because they do not affect the state of the interface.
866 Each of these functions may be issued from any state except the uninitialised state. Similarly,
867 the *t_error()* and *t_strerror()* functions have been excluded from the state table because they do
868 not affect the state of the interface.

869 **5.1 Transport Interfaces States**

870 XTI manages a transport endpoint by using at most 8 states:

- 871 • T_UNINIT
- 872 • T_UNBND
- 873 • T_IDLE
- 874 • T_OUTCON
- 875 • T_INCON
- 876 • T_DATAXFER
- 877 • T_INREL
- 878 • T_OUTREL.

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

881 Table 5-1 describes all possible states of the transport provider as seen by the transport user. The
882 service type may be connection mode, connection mode with orderly release or connectionless
883 mode.
884

State	Description	Service Type
T_UNINIT	uninitialised - 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	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

885 **Table 5-1** Transport Interface States

895

906 **5.2 Outgoing Events**

907 The following outgoing events correspond to the successful return or error return of the
 908 specified user-level transport functions causing XTI to change state, where these functions send
 909 a request or response to the transport provider. In Table 5-2, some events (for example, accept1,
 910 accept2 and accept3) are distinguished by the context in which they occur. The context is based
 911 on the values of the following:

912 *ocnt* Count of outstanding connect indications (connect indications passed to the user
 913 but not accepted or rejected).

914 *fd* File descriptor of the current transport endpoint.

915 *resfd* File descriptor of the transport endpoint where a connection will be accepted.

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

943 **Table 5-2** Transport Interface Outgoing Events

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

945 5.3 Incoming Events

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

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

955 In Table 5-3, the *rcvdis* events are distinguished by the context in which they occur. The context
 956 is based on the value of *ocnt*, which is the count of outstanding connect indications on the
 957 current transport endpoint.

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

978 **Table 5-3** Transport Interface Incoming Events

979 **5.4 Transport User Actions**

980 Some state transitions are accompanied by a list of actions the transport user must take. These
981 actions are represented by the notation [n], where *n* is the number of the specific action as
982 described in Table 5-4.
983

984

- | | |
|-----|---|
| [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 <i>t_accept()</i> . |

985

986

987

988

989

Table 5-4 Transport Interface User Actions

990 5.5 State Tables

991 Table 5-5, Table 5-6 and Table 5-7 describe the possible next states, given the current state and
 992 event. The state is that of the transport provider as seen by the transport user.

993 The contents of each box represent the next state given the current state (column) and the
 994 current incoming or outgoing event (row). An empty box represents a state/event combination
 995 that is invalid. Along with the next state, each box may include an action list (as specified in
 996 Table 5-4 on page 31). The transport user must take the specific actions in the order specified in
 997 the state table.

998 A separate table is shown for initialisation/de-initialisation, data transfer in connectionless
 999 mode and connection/release/data transfer in connection mode.

1000

1001

1002

1003

1004

1005

1006

state event	T_UNINIT	T_UNBND	T_IDLE
opened	T_UNBND		
bind		T_IDLE [1]	
unbind			T_UNBND
closed		T_UNINIT	T_UNINIT

1007

Table 5-5 Initialisation/De-initialisation States

1008

1009

1010

1011

1012

1013

state event	T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

1014

Table 5-6 Data Transfer States: Connectionless-mode Service

1015

1016

1017

1018

1019

1020

1021

1022

1023

1024

1025

1026

1027

1028

1029

1030

1031

1032

1033

1034

1035

state event	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL	T_UNBND
<i>connect1</i>	T_DATAXFER						
<i>connect2</i>	T_OUTCON						
<i>rcvconnect</i>		T_DATAXFER					
<i>listen</i>	T_INCON[2]		T_INCON[2]				
<i>accept1</i>			T_DATAXFER[3]				
<i>accept2</i>			T_IDLE[3][4]				
<i>accept3</i>			T_INCON[3][4]				
<i>snd</i>				T_DATAXFER		T_INREL	
<i>rcv</i>				T_DATAXFER	T_OUTREL		
<i>snddis1</i>		T_IDLE	T_IDLE[3]	T_IDLE	T_IDLE	T_IDLE	
<i>snddis2</i>			T_INCON[3]				
<i>rcvdis1</i>		T_IDLE		T_IDLE	T_IDLE	T_IDLE	
<i>rcvdis2</i>			T_IDLE[3]				
<i>rcvdis3</i>			T_INCON[3]				
<i>sndrel</i>				T_OUTREL		T_IDLE	
<i>rcvrel</i>				T_INREL	T_IDLE		
<i>pass_conn</i>	T_DATAXFER						T_DATAXFER
<i>optgmt</i>	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL	T_UNBIND
<i>closed</i>	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	

1036

Table 5-7 Connection/Release/Data Transfer States: Connection-mode Service

1037 5.6 Events and TLOOK Error Indication

1038 The following list describes the asynchronous events which cause an XTI call to return with a
1039 [TLOOK] error:

1040	<i>t_accept()</i>	T_DISCONNECT, T_LISTEN
1041	<i>t_connect()</i>	T_DISCONNECT, T_LISTEN ¹
1042	<i>t_listen()</i>	T_DISCONNECT ²
1043	<i>t_rcv()</i>	T_DISCONNECT, T_ORDREL ³
1044	<i>t_rcvconnect()</i>	T_DISCONNECT
1045	<i>t_rcvrel()</i>	T_DISCONNECT
1046	<i>t_rcvudata()</i>	T_UDERR
1047	<i>t_snd()</i>	T_DISCONNECT, T_ORDREL
1048	<i>t_sndudata()</i>	T_UDERR
1049	<i>t_unbind()</i>	T_LISTEN, T_DATA ⁴
1050	<i>t_sndrel()</i>	T_DISCONNECT
1051	<i>t_snddis()</i>	T_DISCONNECT

1052 Once a [TLOOK] error has been received on a transport endpoint via an XTI function,
1053 subsequent calls to that and other XTI functions, to which the same [TLOOK] error applies, will
1054 continue to return [TLOOK] until the event is consumed. An event causing the [TLOOK] error
1055 can be determined by calling *t_look()* and then can be consumed by calling the corresponding
1056 consuming XTI function as defined in Table 3-1.

1057 _____

- 1058 1. This occurs only when a *t_connect* is done on an endpoint which has been bound with a *qlen* > 0 and for which a connect
1059 indication is pending.
- 1060 2. This event indicates a disconnect on an outstanding connect indication.
- 1061 3. This occurs only when all pending data has been read.
- 1062 4. T_DATA may only occur for the connectionless mode.

The Use of Options in XTI

1063

1064 6.1 Generalities

1065 The functions `t_accept()`, `t_connect()`, `t_listen()`, `t_optmgmt()`, `t_rcvconnect()`, `t_rcvudata()`,
 1066 `t_rcvuderr()` and `t_sndudata()` contain an *opt* argument of type `struct netbuf` as an input or
 1067 output parameter. This argument is used to convey options between the transport user and the
 1068 transport provider.

1069 There is no general definition about the possible contents of options. There are general XTI
 1070 options and those that are specific for each transport provider. Some options allow the user to
 1071 tailor his communication needs, for instance by asking for high throughput or low delay. Others
 1072 allow the fine-tuning of the protocol behaviour so that communication with unusual
 1073 characteristics can be handled more effectively. Other options are for debugging purposes.

1074 All options have default values. Their values have meaning to and are defined by the protocol
 1075 level in which they apply. However, their values can be negotiated by a transport user. This
 1076 includes the simple case where the transport user can simply enforce its use. Often, the
 1077 transport provider or even the remote transport user may have the right to negotiate a value of
 1078 lesser quality than the proposed one, that is, a delay may become longer, or a throughput may
 1079 become lower.

1080 It is useful to differentiate between options that are *association-related*⁵ and those that are not.
 1081 Association-related options are intimately related to the particular transport connection or
 1082 datagram transmission. If the calling user specifies such an option, some ancillary information is
 1083 transferred across the network in most cases. The interpretation and further processing of this
 1084 information is protocol-dependent. For instance, in an ISO connection-oriented communication,
 1085 the calling user may specify quality-of-service parameters on connection establishment. These
 1086 are first processed and possibly lowered by the local transport provider, then sent to the remote
 1087 transport provider that may degrade them again, and finally conveyed to the called user that
 1088 makes the final selection and transmits the selected values back to the caller.

1089 Options that are not association-related do not contain information destined for the remote
 1090 transport user. Some have purely local relevance, for example, an option that enables
 1091 debugging. Others influence the transmission, for instance the option that sets the IP *time-to-live*
 1092 field, or TCP_NODELAY (see Appendix B on page 199). Local options are negotiated solely
 1093 between the transport user and the local transport provider. The distinction between these two
 1094 categories of options is visible in XTI through the following relationship: on output, the
 1095 functions `t_listen()` and `t_rcvudata()` return association-related options only. The functions
 1096 `t_rcvconnect()` and `t_rcvuderr()` may return options of both categories. On input, options of both
 1097 categories may be specified with `t_accept()` and `t_sndudata()`. The functions `t_connect()` and
 1098 `t_optmgmt()` can process and return both categories of options.

1099 The transport provider has a default value for each option it supports. These defaults are
 1100 sufficient for the majority of communication relations. Hence, a transport user should only
 1101 request options actually needed to perform the task, and leave all others at their default value.

1102 _____
 1103 5. The term "association" is used to denote a pair of communicating transport users.

1104 This chapter describes the general framework for the use of options. This framework is
 1105 obligatory for all transport providers. The specific options that are legal for use with a specific
 1106 transport provider are described in the provider-specific appendices (see Appendix A on page
 1107 189 and Appendix B on page 199). General XTI options are described in *t_optmgmt()* on page 76.

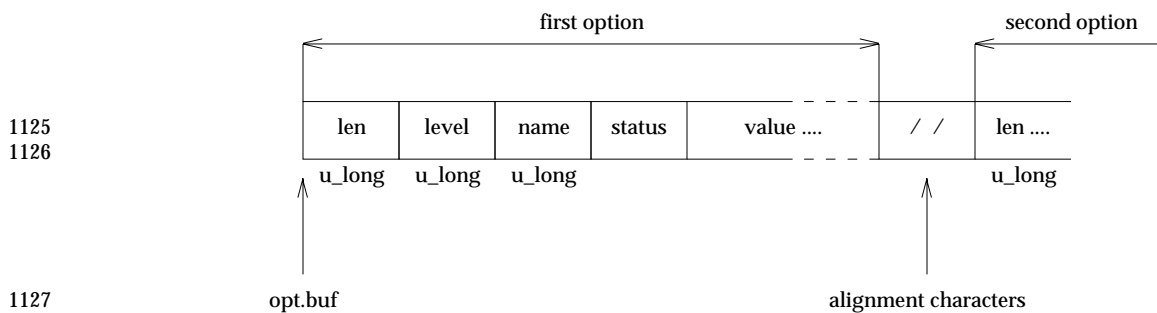
1108 **6.2 The Format of Options**

1109 Options are conveyed via an *opt* argument of **struct netbuf**. Each option in the buffer specified is
 1110 of the form **struct t_opthdr** possibly followed by an option value.

1111 A transport provider embodies a stack of protocols. The *level* field of **struct t_opthdr** identifies
 1112 the XTI level or a protocol of the transport provider as TCP or ISO 8073:1986. The *name* field
 1113 identifies the option within the level, and *len* contains its total length, that is, the length of the
 1114 option header **t_opthdr** plus the length of the option value. The *status* field is used by the XTI
 1115 level or the transport provider to indicate success or failure of a negotiation (see Section 6.3.5 on
 1116 page 40 and *t_optmgmt()* on page 76).

1117 Several options can be concatenated. The transport user has, however, to ensure that each
 1118 option starts at a long-word boundary. The macro **OPT_NEXTHDR(pbuf, buflen, poption)** can
 1119 be used for that purpose. The parameter *pbuf* denotes a pointer to an option buffer *opt.buf*, and
 1120 *buflen* is its length. The parameter *poption* points to the current option in the option buffer.
 1121 **OPT_NEXTHDR** returns a pointer to the position of the next option, or returns a null pointer if
 1122 the option buffer is exhausted. The macro is helpful for writing and reading. See <**xti.h**> in
 1123 Appendix F on page 253 for the exact definition.

1124 The option buffer thus has the following form (unsigned long is abbreviated to *u_long*):



1128 The length of the option buffer is given by *opt.len*.

1129 6.3 The Elements of Negotiation

1130 This section describes the general rules governing the passing and retrieving of options and the
 1131 error conditions that can occur. Unless explicitly restricted, these rules apply to all functions
 1132 that allow the exchange of options.

1133 6.3.1 Multiple Options and Options Levels

1134 When multiple options are specified in an option buffer on input, different rules apply to the
 1135 levels that may be specified, depending on the function call. Multiple options specified on input
 1136 to `t_optmgmt()` must address the same option level. Options specified on input to `t_connect()`,
 1137 `t_accept()` and `t_sndudata()` can address different levels.

1138 6.3.2 Illegal Options

1139 Only legal options can be negotiated; illegal options cause failure. An option is illegal if the
 1140 following applies:

- 1141 • The length specified in `t_opthdr.len` exceeds the remaining size of the option buffer (counted
 1142 from the beginning of the option).
- 1143 • The option value is illegal. The legal values are defined for each option. (See `t_optmgmt()` on
 1144 page 76, Appendix A on page 189 and Appendix B on page 199.)

1145 If an illegal option is passed to XTI, the following will happen:

- 1146 • A call to `t_optmgmt()` fails with [TBADOPT].
- 1147 • `t_accept()` or `t_connect()` fail either with [TBADOPT], or the connection establishment aborts,
 1148 depending on the implementation and the time the illegal option is detected. If the
 1149 connection aborts, a T_DISCONNECT event occurs, and a synchronous call to `t_connect()`
 1150 fails with [TLOOK]. It depends on timing and implementation conditions whether a
 1151 `t_accept()` call still succeeds or fails with [TLOOK] in that case.
- 1152 • A call to `t_sndudata()` either fails with [TBADOPT], or it successfully returns, but a T_UDERR
 1153 event occurs to indicate that the datagram was not sent.

1154 If the transport user passes multiple options in one call and one of them is illegal, the call fails as
 1155 described above. It is, however, possible that some or even all of the submitted legal options
 1156 were successfully negotiated. The transport user can check the current status by a call to
 1157 `t_optmgmt()` with the T_CURRENT flag set (see `t_optmgmt()` on page 76).

1158 Specifying an option level unknown to the transport provider does not cause failure in calls to
 1159 `t_connect()`, `t_accept()` or `t_sndudata()`; the option is discarded in these cases. The function
 1160 `t_optmgmt()` fails with [TBADOPT].

1161 Specifying an option name that is unknown to or not supported by the protocol selected by the
 1162 option level does not cause failure. The option is discarded in calls to `t_connect()`, `t_accept()` or
 1163 `t_sndudata()`. The function `t_optmgmt()` returns T_NOTSUPPORT in the `level` field of the option.

1164 6.3.3 Initiating an Option Negotiation

1165 A transport user initiates an option negotiation when calling *t_connect()*, *t_sndudata()* or
1166 *t_optmgmt()* with the flag T_NEGOTIATE set.

1167 The negotiation rules for these functions depend on whether an option request is an absolute
1168 requirement or not. This is explicitly defined for each option (see *t_optmgmt()* on page 76,
1169 Appendix A on page 189 and Appendix B on page 199). In case of an ISO transport provider, for
1170 example, the option that requests use of expedited data is not an absolute requirement. On the
1171 other hand, the option that requests protection could be an absolute requirement.

1172 **Note:** The notion “absolute requirement” originates from the quality-of-service parameters in
1173 ISO 8072:1986. Its use is extended here to all options.

1174 If the proposed option value is an absolute requirement, three outcomes are possible:

- 1175 • The negotiated value is the same as the proposed one. When the result of the negotiation is
1176 retrieved, the *status* field in **t_opthdr** is set to T_SUCCESS.
- 1177 • The negotiation is rejected if the option is supported but the proposed value cannot be
1178 negotiated. This leads to the following behaviour:
 - 1179 — *t_optmgmt()* successfully returns, but the returned option has its *status* field set to
1180 T_FAILURE.
 - 1181 — Any attempt to establish a connection aborts; a T_DISCONNECT event occurs, and a
1182 synchronous call to *t_connect()* fails with [TLOOK].
 - 1183 — *t_sndudata()* fails with [TLOOK] or successfully returns, but a T_UDERR event occurs to
1184 indicate that the datagram was not sent.

1185 If multiple options are submitted in one call and one of them is rejected, XTI behaves as just
1186 described. Although the connection establishment or the datagram transmission fails,
1187 options successfully negotiated before some option was rejected retain their negotiated
1188 values. There is no roll-back mechanism (see Section 6.4 on page 42).

1189 The function *t_optmgmt()* attempts to negotiate each option. The *status* fields of the returned
1190 options indicate success (T_SUCCESS) or failure (T_FAILURE).

- 1191 • If the local transport provider does not support the option at all, *t_optmgmt()* reports
1192 T_NOTSUPPORT in the *status* field. The functions *t_connect()* and *t_sndudata()* ignore this
1193 option.

1194 If the proposed option value is not an absolute requirement, two outcomes are possible:

- 1195 • The negotiated value is of equal or lesser quality than the proposed one (for example, a delay
1196 may become longer).
 - 1197 When the result of the negotiation is retrieved, the *status* field in **t_opthdr** is set to
1198 T_SUCCESS if the negotiated value equals the proposed one, or set to T_PARTSUCCESS
1199 otherwise.
- 1200 • If the local transport provider does not support the option at all, *t_optmgmt()* reports
1201 T_NOTSUPPORT in the *status* field. The functions *t_connect()* and *t_sndudata()* ignore this
1202 option.

1203 Unsupported options do not cause functions to fail or a connection to abort, since different
1204 vendors possibly implement different subsets of options. Furthermore, future enhancements of
1205 XTI might encompass additional options that are unknown to earlier implementations of
1206 transport providers. The decision whether or not the missing support of an option is acceptable
1207 for the communication is left to the transport user.

1208 The transport provider does not check for multiple occurrences of the same option, possibly
 1209 with different option values. It simply processes the options in the option buffer one after the
 1210 other. However, the user should not make any assumption about the order of processing.

1211 Not all options are independent of one another. A requested option value might conflict with
 1212 the value of another option that was specified in the same call or is currently effective (see
 1213 Section 6.4 on page 42). These conflicts may not be detected at once, but later they might lead to
 1214 unpredictable results. If detected at negotiation time, these conflicts are resolved within the
 1215 rules stated above. The outcomes may thus be quite different and depend on whether absolute
 1216 or non-absolute requests are involved in the conflict.

1217 Conflicts are usually detected at the time a connection is established or a datagram is sent. If
 1218 options are negotiated with *t_optmgmt()*, conflicts are usually not detected at this time, since
 1219 independent processing of the requested options must allow for temporal inconsistencies.

1220 When called, the functions *t_connect()* and *t_sndudata()* initiate a negotiation of *all* association-
 1221 related options according to the rules of this section. Options not explicitly specified in the
 1222 function calls themselves are taken from an internal option buffer that contains the values of a
 1223 previous negotiation (see Section 6.4 on page 42).

1224 **6.3.4 Responding to a Negotiation Proposal**

1225 In connection-oriented communication, some protocols give the peer transport users the
 1226 opportunity to negotiate characteristics of the transport connection to be established. These
 1227 characteristics are association-related options. With the connect indication, the called user
 1228 receives (via *t_listen()*) a proposal about the option values that should be effective for this
 1229 connection. The called user can accept this proposal or weaken it by choosing values of lower
 1230 quality (for example, longer delays than proposed). The called user can, of course, refuse the
 1231 connection establishment altogether.

1232 The called user responds to a negotiation proposal via *t_accept()*. If the called transport user
 1233 tries to negotiate an option of higher quality than proposed, the outcome depends on the
 1234 protocol to which that option applies. Some protocols may reject the option, some protocols
 1235 take other appropriate action described in protocol-specific appendices. If an option is rejected,
 1236 the following error occurs:

1237 The connection fails; a T_DISCONNECT event occurs. It depends on timing and
 1238 implementation conditions whether the *t_accept()* call still succeeds or fails with
 1239 [TLOOK].

1240 If multiple options are submitted with *t_accept()* and one of them is rejected, the connection fails
 1241 as described above. Options that could be successfully negotiated before the erroneous option
 1242 was processed retain their negotiated value. There is no roll-back mechanism (see Section 6.4 on
 1243 page 42).

1244 The response options can either be specified with the *t_accept()* call, or can be preset for the
 1245 responding endpoint (not the listening endpoint!) *resfd* in a *t_optmgmt()* call (action
 1246 T_NEGOTIATE) prior to *t_accept()* (see Section 6.4 on page 42). Note that the response to a
 1247 negotiation proposal is activated when *t_accept()* is called. A *t_optmgmt()* call with erroneous
 1248 option values as described above will succeed; the connection aborts at the time *t_accept()* is
 1249 called.

1250 The connection also fails if the selected option values lead to contradictions.

1251 The function *t_accept()* does not check for multiple specification of an option (see Section 6.3.3
 1252 on page 38). Unsupported options are ignored.

1253 **6.3.5 Retrieving Information about Options**

1254 This section describes how a transport user can retrieve information about options. To be
 1255 explicit, a transport user must be able to:

- 1256 • know the result of a negotiation (for example, at the end of a connection establishment)
- 1257 • know the proposed option values under negotiation (during connection establishment)
- 1258 • retrieve option values sent by the remote transport user for notification only (for example, IP
 1259 options)
- 1260 • check option values currently effective for the transport endpoint.

1261 To this end, the functions *t_connect()*, *t_listen()*, *t_optmgmt()*, *t_rcvconnect()*, *t_rcvudata()* and
 1262 *t_rcvuderr()* take an output argument *opt* of **struct netbuf**. The transport user has to supply a
 1263 buffer where the options shall be written to; *opt.buf* must point to this buffer, and *opt.maxlen*
 1264 must contain the buffer's size. The transport user can set *opt.maxlen* to zero to indicate that no
 1265 options are to be retrieved.

1266 Which options are returned depend on the function call involved:

1267 *t_connect()* (synchronous mode) and *t_rcvconnect()*

1268 The functions return the values of all association-related options that were
 1269 received with the connection response and the negotiated values of those non-
 1270 association-related options that had been specified on input. However, options
 1271 specified on input in the *t_connect()* call that are not supported or refer to an
 1272 unknown option level are discarded and not returned on output.

1273 The *status* field of each option returned with *t_connect()* or *t_rcvconnect()* indicates
 1274 if the proposed value (T_SUCCESS) or a degraded value (T_PARTSUCCESS) has
 1275 been negotiated. The *status* field of received ancillary information (for example, IP
 1276 options) that is not subject to negotiation is always set to T_SUCCESS.

1277 *t_listen()* The received association-related options are related to the incoming connection
 1278 (identified by the sequence number), not to the listening endpoint. (However, the
 1279 option values currently effective for the listening endpoint can affect the values
 1280 retrieved by *t_listen()*, since the transport provider might be involved in the
 1281 negotiation process, too.) Thus, if the same options are specified in a call to
 1282 *t_optmgmt()* with action T_CURRENT, will usually not return the same values.

1283 The number of received options may be variable for subsequent connect
 1284 indications, since many association-related options are only transmitted on explicit
 1285 demand by the calling user (for example, IP options or ISO 8072:1986 throughput).
 1286 It is even possible that no options at all are returned.

1287 The *status* field is irrelevant.

1288 *t_rcvudata()* The received association-related options are related to the incoming datagram, not
 1289 to the transport endpoint *fd*. Thus, if the same options are specified in a call to
 1290 *t_optmgmt()* with action T_CURRENT, *t_optmgmt()* will usually not return the
 1291 same values.

1292 The number of options received may vary from call to call.

1293 The *status* field is irrelevant.

1294 *t_rcvuderr()* The returned options are related to the options input at the previous *t_sndudata()*
 1295 call that produced the error. Which options are returned and which values they
 1296 have depend on the specific error condition.

1297 The *status* field is irrelevant.

1298 *t_optmgmt()* This call can process and return both categories of options. It acts on options
 1299 related to the specified transport endpoint, not on options related to a connect
 1300 indication or an incoming datagram. A detailed description is given in
 1301 *t_optmgmt()* on page 76.

1302 6.3.6 Privileged and Read-only Options

1303 *Privileged* options or option values are those that may be requested by privileged users only. The
 1304 meaning of privilege is hereby implementation-defined.

1305 *Read-only* options serve for information purposes only. The transport user may be allowed to
 1306 read the option value but not to change it. For instance, to select the value of a protocol timer or
 1307 the maximum length of a protocol data unit may be too subtle to leave to the transport user,
 1308 though the knowledge about this value might be of some interest. An option might be read-only
 1309 for all users or solely for non-privileged users. A privileged option might be inaccessible or
 1310 read-only for non-privileged users.

1311 An option might be negotiable in some XTI states and read-only in other XTI states. For
 1312 instance, the ISO quality-of-service options are negotiable in the states T_IDLE and T_INCON
 1313 and read-only in all other states (except T_UNINIT).

1314 If a transport user requests negotiation of a read-only option, or a non-privileged user requests
 1315 illegal access to a privileged option, the following outcomes are possible:

- 1316 • *t_optmgmt()* successfully returns, but the returned option has its *status* field set to
 1317 T_NOTSUPPORT if a privileged option was requested illegally, and to T_READONLY if
 1318 modification of a read-only option was requested.
- 1319 • If negotiation of a read-only option is requested, *t_accept()* or *t_connect()* either fail with
 1320 [TACCES], or the connection establishment aborts and a T_DISCONNECT event occurs. If
 1321 the connection aborts, a synchronous call to *t_connect()* fails with [TLOOK]. If a privileged
 1322 option is illegally requested, the option is quietly ignored. (A non-privileged user shall not
 1323 be able to select an option which is privileged or unsupported.) It depends on timing and
 1324 implementation conditions whether a *t_accept()* call still succeeds or fails with [TLOOK].
- 1325 • If negotiation of a read-only option is requested, *t_sndudata()* may return [TLOOK] or
 1326 successfully return, but a T_UDERR event occurs to indicate that the datagram was not sent.
 1327 If a privileged option is illegally requested, the option is quietly ignored. (A non-privileged
 1328 user shall not be able to select an option which is privileged or unsupported.)

1329 If multiple options are submitted to *t_connect()*, *t_accept()* or *t_sndudata()* and a read-only
 1330 option is rejected, the connection or the datagram transmission fails as described. Options that
 1331 could be successfully negotiated before the erroneous option was processed retain their
 1332 negotiated values. There is no roll-back mechanism (see also Section 6.4 on page 42).

1333 6.4 Option Management of a Transport Endpoint

1334 This section describes how option management works during the lifetime of a transport
1335 endpoint.

1336 Each transport endpoint is (logically) associated with an internal option buffer. When a
1337 transport endpoint is created, this buffer is filled with a system default value for each supported
1338 option. Depending on the option, the default may be 'OPTION ENABLED', 'OPTION
1339 DISABLED' or denote a time span, etc. These default settings are appropriate for most uses.
1340 Whenever an option value is modified in the course of an option negotiation, the modified value
1341 is written to this buffer and overwrites the previous one. At any time, the buffer contains all
1342 option values that are currently effective for this transport endpoint.

1343 The current value of an option can be retrieved at any time by calling *t_optmgmt()* with the flag
1344 T_CURRENT set. Calling *t_optmgmt()* with the flag T_DEFAULT set yields the system default
1345 for the specified option.

1346 A transport user can negotiate new option values by calling *t_optmgmt()* with the flag
1347 T_NEGOTIATE set. The negotiation follows the rules described in Section 6.3 on page 37.

1348 Some options may be modified only in specific XTI states and are read-only in other XTI states.
1349 Many association-related options, for instance, may not be changed in the state T_DATAXFER,
1350 and an attempt to do so will fail (see Section 6.3.6 on page 41). The legal states for each option
1351 are specified with its definition.

1352 As usual, association-related options take effect at the time a connection is established or a
1353 datagram is transmitted. This is the case if they contain information that is transmitted across
1354 the network or determine specific transmission characteristics. If such an option is modified by
1355 a call to *t_optmgmt()*, the transport provider checks whether the option is supported and
1356 negotiates a value according to its current knowledge. This value is written to the internal
1357 option buffer.

1358 The final negotiation takes place if the connection is established or the datagram is transmitted.
1359 This can result in a degradation of the option value or even in a negotiation failure. The
1360 negotiated values are written to the internal option buffer.

1361 Some options may be changed in the state T_DATAXFER, for example, those specifying buffer
1362 sizes. Such changes might affect the transmission characteristics and lead to unexpected side
1363 effects (for example, data loss if a buffer size was shortened) if the user does not care.

1364 The transport user can explicitly specify both categories of options on input when calling
1365 *t_connect()*, *t_accept()* or *t_sndudata()*. The options are at first locally negotiated option-by-
1366 option, and the resulting values written to the internal option buffer. The modified option buffer
1367 is then used if a further negotiation step across the network is required, as for instance in
1368 connection-oriented ISO communication. The newly negotiated values are then written to the
1369 internal option buffer.

1370 At any stage, a negotiation failure can lead to an abort of the transmission. If a transmission
1371 aborts, the option buffer will preserve the content it had at the time the failure occurred.
1372 Options that could be negotiated just before the error occurred are written back to the option
1373 buffer, whether the XTI call fails or succeeds.

1374 It is up to the transport user to decide which options it explicitly specifies on input when calling
1375 *t_connect()*, *t_accept()* or *t_sndudata()*. The transport user need not pass options at all, by setting
1376 the *len* field of the function's input *opt* argument to zero. The current content of the internal
1377 option buffer is then used for negotiation without prior modification.

1378 The negotiation procedure for options at the time of a `t_connect()`, `t_accept()` or `t_sndudata()` call
1379 always obeys the rules in Section 6.3.3 on page 38 and Section 6.3.4 on page 39, whether the
1380 options were explicitly specified during the call or implicitly taken from the internal option
1381 buffer.

1382 The transport user should not make assumptions about the order in which options are processed
1383 during negotiation.

1384 A value in the option buffer is only modified as a result of a successful negotiation of this option.
1385 It is, in particular, not changed by a connection release. There is no history mechanism that
1386 would restore the buffer state existing prior to the connection establishment or the datagram
1387 transmission. The transport user must be aware that a connection establishment or a datagram
1388 transmission may change the internal option buffer, even if each option was originally initialised
1389 to its default value.

1390 6.5 Supplements

1391 This section contains supplementary remarks and a short summary.

1392 6.5.1 The Option Value T_UNSPEC

1393 Some options may not have a fully specified value all the time. An ISO transport provider, for
1394 instance, that supports several protocol classes, might not have a preselected preferred class
1395 before a connection establishment is initiated. At the time of the connection request, the
1396 transport provider may conclude from the destination address, quality-of-service parameters
1397 and other locally available information which preferred class it should use. A transport user
1398 asking for the default value of the preferred class option in state T_IDLE would get the value
1399 T_UNSPEC. This value indicates that the transport provider did not yet select a value. The
1400 transport user could negotiate another value as the preferred class, for example, T_CLASS2. The
1401 transport provider would then be forced to initiate a connect request with class 2 as the
1402 preferred class.

1403 An XTI implementation may also return the value T_UNSPEC if it can currently not access the
1404 option value. This may happen, for example, in the state T_UNBND in systems where the
1405 protocol stacks reside on separate controller cards and not in the host. The implementation may
1406 never return T_UNSPEC if the option is not supported at all.

1407 If T_UNSPEC is a legal value for a specific option, it may be used by the user on input, too. It is
1408 used to indicate that it is left to the provider to choose an appropriate value. This is especially
1409 useful in complex options as ISO throughput, where the option value has an internal structure
1410 (see TCO_THROUGHPUT in Appendix A on page 189). The transport user may leave some
1411 fields unspecified by selecting this value. If the user proposes T_UNSPEC, the transport
1412 provider is free to select an appropriate value. This might be the default value, some other
1413 explicit value, or T_UNSPEC.

1414 For each option, it is specified whether or not T_UNSPEC is a legal value for negotiation
1415 purposes.

1416 6.5.2 The info Argument

1417 The functions *t_open()* and *t_getinfo()* return values representing characteristics of the transport
1418 provider in the argument *info*. The value of *info->options* is used by *t_alloc()* to allocate storage
1419 for an option buffer to be used in an XTI call. The value is sufficient for all uses.

1420 In general, *info->options* also includes the size of privileged options, even if these are not read-
1421 only for non-privileged users. Alternatively, an implementation can choose to return different
1422 values in *info->options* for privileged and non-privileged users.

1423 The values in *info->etsdu*, *info->tsdu*, *info->connect* and *info->discon* possibly diminish as soon as
1424 the T_DATAXFER state is entered. Calling *t_optmgmt()* does not influence these values (see
1425 *t_optmgmt()* on page 76).

1426 **6.5.3 Summary**

- 1427 • The format of an option is defined by a header **struct t_opthdr**, followed by an option value.
- 1428 • On input, several options can be specified in an input *opt* argument. Each option must begin
1429 on a long-word boundary.
- 1430 • There are options that are association-related and options that are not. On output, the
1431 functions *t_listen()* and *t_rcvudata()* return association-related options only. The functions
1432 *t_rcvconnect()* and *t_rcvuderr()* may return options of both categories. On input, options of
1433 both categories may be specified with *t_accept()* and *t_sndudata()*. The functions *t_connect()*
1434 and *t_optmgmt()* can process and return both categories of options.
- 1435 • A transport endpoint is (logically) associated with an internal option buffer, where the
1436 currently effective values are stored. Each successful negotiation of an option modifies this
1437 buffer, regardless of whether the call initiating the negotiation succeeds or fails.
- 1438 • When calling *t_connect()*, *t_accept()* or *t_sndudata()*, the transport user can choose to submit
1439 the currently effective option values by setting the *len* field of the input *opt* argument to zero.
- 1440 • If a connection is accepted via *t_accept()*, the explicitly specified option values together with
1441 the currently effective option values of *resfd*, not of *fd*, matter in this negotiation step.
- 1442 • The options returned by *t_rcvuderr()* are those negotiated with the outgoing datagram that
1443 produced the error. If the error occurred during option negotiation, the returned option
1444 might represent some mixture of partly negotiated and not-yet negotiated options.

1445 6.6 Portability Aspects

1446 An application programmer who writes XTI programs faces two portability aspects:

- 1447 • portability across protocol profiles
- 1448 • portability across different system platforms (possibly from different vendors).

1449 Options are intrinsically coupled with a definite protocol or protocol profile. Making explicit
1450 use of them therefore degrades portability across protocol profiles.

1451 Different vendors might offer transport providers with different option support. This is due to
1452 different implementations and product policies. The lists of options on the *t_optmgmt()* manual
1453 page and in the protocol-specific appendices are maximal sets but do not necessarily reflect
1454 common implementation practice. Vendors will implement subsets that suit their needs.
1455 Making careless use of options therefore endangers portability across different system
1456 platforms.

1457 Every implementation of a protocol profile accessible by XTI can be used with the default values
1458 of options. Applications can thus be written that do not care about options at all.

1459 An application program that processes options retrieved from an XTI function should discard
1460 options it does not know in order to lessen its dependence from different system platforms and
1461 future XTI releases with possibly increased option support.

XTI Library Functions and Parameters

1462

1463 7.1 How to Prepare XTI Applications

1464 In a software development environment, a program, for example that uses XTI functions must
 1465 be compiled with the XTI Library. This can be done using the following command (for example,
 1466 for normal library):

```
1467     cc file.c -lxti
```

1468 The syntax for shared libraries is implementation-dependent.

1469 The XTI structures and constants are all defined in the `<xti.h>` header, which can be found in
 1470 Appendix F on page 253.

1471 7.2 Key for Parameter Arrays

1472 For each XTI function description, a table is given which summarises the contents of the input
 1473 and output parameter. The key is given below:

1474 x The parameter value is meaningful. (Input parameter must be set before the call and
 1475 output parameter may be read after the call.)

1476 (x) The content of the object pointed to by the x pointer is meaningful.

1477 ? The parameter value is meaningful but the parameter is optional.

1478 (?) The content of the object pointed to by the ? pointer is optional.

1479 / The parameter value is meaningless.

1480 = The parameter after the call keeps the same value as before the call.

1481 7.3 Return of TLOOK Error

1482 Many of the XTI functions contained in this chapter return a [TLOOK] error to report the
 1483 occurrence of an asynchronous event. For these functions a complete list describing the function
 1484 and the events is provided in Section 5.6 on page 34.

1485 NAME

1486 t_accept - accept a connect request

1487 SYNOPSIS

1488 #include <xti.h>

1489 int t_accept(int *fd*, int *resfd*, struct t_call **call*);

1490 DESCRIPTION

1491

1492

1493

1494

1495

1496

1497

1498

1499

1500

1501

1502

1503

1504

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

1505

1506

1507

1508

1509

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

1510

1511

1512

1513

```

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

```

1514

1515

1516

1517

1518

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

1519

1520

1521

1522

1523

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

1524

1525

1526

1527

1528

If a different transport endpoint is specified (*resfd*!=*fd*), then the user may or may not choose to bind the endpoint before the *t_accept()* is issued. If the endpoint is not bound prior to the *t_accept()*, then the transport provider will automatically bind it to the same protocol address *fd* is bound to. If the transport user chooses to bind the endpoint it must be bound to a protocol address with a *qlen* of zero and must be in the T_IDLE state before the *t_accept()* is issued.

1529

1530

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

1531 The *udata* argument enables the called transport user to send user data to the caller and the
 1532 amount of user data must not exceed the limits supported by the transport provider as returned
 1533 in the *connect* field of the *info* argument of *t_open()* or *t_getinfo()*. If the *len* field of *udata* is zero,
 1534 no data will be sent to the caller. All the *maxlen* fields are meaningless.

1535 When the user does not indicate any option (call->opt.len = 0) it is assumed that the connection
 1536 is to be accepted unconditionally. The transport provider may choose options other than the
 1537 defaults to ensure that the connection is accepted successfully.

1538 CAVEATS

1539 There may be transport provider-specific restrictions on address binding. See Appendix A on
 1540 page 189 and Appendix B on page 199.

1541 Some transport providers do not differentiate between a connect indication and the connection
 1542 itself. If the connection has already been established after a successful return of *t_listen()*,
 1543 *t_accept()* will assign the existing connection to the transport endpoint specified by *resfd* (see
 1544 Appendix B on page 199).

1545 VALID STATES

1546 fd: T_INCON resfd (fd!=resfd): T_IDLE

1547 ERRORS

1548 On failure, *t_errno* is set to one of the following:

1549	[TBADF]	The file descriptor <i>fd</i> or <i>resfd</i> does not refer to a transport endpoint.
1550	[TOUTSTATE]	The function was called in the wrong sequence on the transport endpoint 1551 referenced by <i>fd</i> , or the transport endpoint referred to by <i>resfd</i> is not in the 1552 appropriate state.
1553	[TACCES]	The user does not have permission to accept a connection on the 1554 responding transport endpoint or to use the specified options.
1555	[TBADOPT]	The specified options were in an incorrect format or contained illegal 1556 information.
1557	[TBADDATA]	The amount of user data specified was not within the bounds allowed by 1558 the transport provider.
1559	[TBADADDR]	The specified protocol address was in an incorrect format or contained 1560 illegal information.
1561	[TBADSEQ]	An invalid sequence number was specified.
1562	[TLOOK]	An asynchronous event has occurred on the transport endpoint 1563 referenced by <i>fd</i> and requires immediate attention.
1564	[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
1565	[TSYSERR]	A system error has occurred during execution of this function.
1566	[TINDOUT]	The function was called with <i>fd==resfd</i> but there are outstanding 1567 connection indications on the endpoint. Those other connection 1568 indications must be handled either by rejecting them via <i>t_snddis(3)</i> or 1569 accepting them on a different endpoint via <i>t_accept(3)</i> .
1570	[TPRIVMISMATCH]	The file descriptors <i>fd</i> and <i>resfd</i> do not refer to the same transport 1571 provider.
1572	[TRESQLEN]	The endpoint referenced by <i>resfd</i> (where <i>resfd != fd</i>) was bound to a 1573 protocol address with a <i>qlen</i> that is greater than zero.

1574 [TPROTO] This error indicates that a communication problem has been detected
1575 between XTI and the transport provider for which there is no other
1576 suitable XTI (*t_errno*).

1577 [TRESADDR] This transport provider requires both *fd* and *resfd* to be bound to the same
1578 address. This error results if they are not.

1579 **RETURN VALUE**

1580 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
1581 *t_errno* is set to indicate an error.

1582 **SEE ALSO**

1583 *t_connect()*, *t_getstate()*, *t_listen()*, *t_open()*, *t_optmgmt()*, *t_rcvconnect()*.

1584 **CHANGE HISTORY**

1585 **Issue 4**

1586 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

1587 **NAME**

1588 t_alloc - allocate a library structure

1589 **SYNOPSIS**

1590 #include <xti.h>

1591 char *t_alloc(int *fd*, int *struct_type*, int *fields*);1592 **DESCRIPTION**

1593

1594

Parameters	Before call	After call
<i>fd</i>	x	/
<i>struct_type</i>	x	/
<i>fields</i>	x	/

1595

1596

1597

1598 The *t_alloc()* function dynamically allocates memory for the various transport function
 1599 argument structures as specified below. This function will allocate memory for the specified
 1600 structure, and will also allocate memory for buffers referenced by the structure.

1601 The structure to allocate is specified by *struct_type* and must be one of the following:

1602	T_BIND	struct	t_bind
1603	T_CALL	struct	t_call
1604	T_OPTMGMT	struct	t_optmgmt
1605	T_DIS	struct	t_discon
1606	T_UNITDATA	struct	t_unitdata
1607	T_UDERROR	struct	t_uderr
1608	T_INFO	struct	t_info

1609 where each of these structures may subsequently be used as an argument to one or more
 1610 transport functions.

1611 Each of the above structures, except T_INFO, contains at least one field of type **struct netbuf**.
 1612 For each field of this type, the user may specify that the buffer for that field should be allocated
 1613 as well. The length of the buffer allocated will be equal to or greater than the appropriate size as
 1614 returned in the *info* argument of *t_open()* or *t_getinfo()*. The relevant fields of the *info* argument
 1615 are described in the following list. The *fields* argument specifies which buffers to allocate, where
 1616 the argument is the bitwise-or of any of the following:

1617	T_ADDR	The <i>addr</i> field of the t_bind , t_call , t_unitdata or t_uderr structures.
1618	T_OPT	The <i>opt</i> field of the t_optmgmt , t_call , t_unitdata or t_uderr structures.
1619	T_UDATA	The <i>udata</i> field of the t_call , t_discon or t_unitdata structures.
1620	T_ALL	All relevant fields of the given structure. Fields which are not supported 1621 by the transport provider specified by <i>fd</i> will not be allocated.

1622 For each relevant field specified in *fields*, *t_alloc()* will allocate memory for the buffer associated
 1623 with the field, and initialise the *len* field to zero and the *buf* pointer and *maxlen* field accordingly.
 1624 Irrelevant or unknown values passed in *fields* are ignored. Since the length of the buffer
 1625 allocated will be based on the same size information that is returned to the user on a call to
 1626 *t_open()* and *t_getinfo()*, *fd* must refer to the transport endpoint through which the newly
 1627 allocated structure will be passed. In this way the appropriate size information can be accessed.
 1628 If the size value associated with any specified field is -1 or -2 (see *t_open()* or *t_getinfo()*),
 1629 *t_alloc()* will be unable to determine the size of the buffer to allocate and will fail, setting *t_errno*
 1630 to [TSYSERR] and *errno* to [EINVAL]. For any field not specified in *fields*, *buf* will be set to the
 1631 null pointer and *len* and *maxlen* will be set to zero.

1632 Use of *t_alloc()* to allocate structures will help ensure the compatibility of user programs with
1633 future releases of the transport interface functions.

1634 VALID STATES

1635 ALL - apart from T_UNINIT

1636 ERRORS

1637 On failure, *t_errno* is set to one of the following:

1638 [TBADF] The specified file descriptor does not refer to a transport endpoint.

1639 [TSYSERR] A system error has occurred during execution of this function.

1640 [TNOSTRUCTYPE] Unsupported *struct_type* requested. This can include a request for a
1641 structure type which is inconsistent with the transport provider type
1642 specified, that is, connection-oriented or connectionless.

1643 [TPROTO] This error indicates that a communication problem has been detected
1644 between XTI and the transport provider for which there is no other
1645 suitable XTI (*t_errno*).

1646 RETURN VALUE

1647 On successful completion, *t_alloc()* returns a pointer to the newly allocated structure. On
1648 failure, a null pointer is returned.

1649 SEE ALSO

1650 *t_free()*, *t_getinfo()*, *t_open()*.

1651 CHANGE HISTORY**1652 Issue 4**

1653 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

1654 **NAME**

1655 t_bind - bind an address to a transport endpoint

1656 **SYNOPSIS**

1657 #include <xti.h>

1658 int t_bind(int *fd*, struct t_bind **req*, struct t_bind **ret*);1659 **DESCRIPTION**

1660

1661

1662

1663

1664

1665

1666

1667

1668

1669

1670

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

1671

1672

1673

1674

1675

This function associates a protocol address with the transport endpoint specified by *fd* and activates that transport endpoint. In connection mode, the transport provider may begin enqueueing 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.

1676

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

1677

```
struct netbuf addr;
```

1678

```
unsigned qlen;
```

1679

1680

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.

1681

1682

1683

1684

1685

1686

1687

1688

1689

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

1690

1691

1692

1693

1694

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

1695

1696

1697

1698

1699

1700

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

1701 The *qlen* field has meaning only when initialising a connection-mode service. It specifies the
 1702 number of outstanding connect indications that the transport provider should support for the
 1703 given transport endpoint. An outstanding connect indication is one that has been passed to the
 1704 transport user by the transport provider but which has not been accepted or rejected. A value of
 1705 *qlen* greater than zero is only meaningful when issued by a passive transport user that expects
 1706 other users to call it. The value of *qlen* will be negotiated by the transport provider and may be
 1707 changed if the transport provider cannot support the specified number of outstanding connect
 1708 indications. However, this value of *qlen* will never be negotiated from a requested value greater
 1709 than zero to zero. This is a requirement on transport providers; see **CAVEATS** below. On
 1710 return, the *qlen* field in *ret* will contain the negotiated value.

1711 If *fd* refers to a connection-mode service, this function allows more than one transport endpoint
 1712 to be bound to the same protocol address (however, the transport provider must also support
 1713 this capability), but it is not possible to bind more than one protocol address to the same
 1714 transport endpoint. If a user binds more than one transport endpoint to the same protocol
 1715 address, only one endpoint can be used to listen for connect indications associated with that
 1716 protocol address. In other words, only one *t_bind()* for a given protocol address may specify a
 1717 value of *qlen* greater than zero. In this way, the transport provider can identify which transport
 1718 endpoint should be notified of an incoming connect indication. If a user attempts to bind a
 1719 protocol address to a second transport endpoint with a value of *qlen* greater than zero, *t_bind()*
 1720 will return -1 and set *t_errno* to [TADDRBUSY]. When a user accepts a connection on the
 1721 transport endpoint that is being used as the listening endpoint, the bound protocol address will
 1722 be found to be busy for the duration of the connection, until a *t_unbind()* or *t_close()* call has
 1723 been issued. No other transport endpoints may be bound for listening on that same protocol
 1724 address while that initial listening endpoint is active (in the data transfer phase or in the T_IDLE
 1725 state). This will prevent more than one transport endpoint bound to the same protocol address
 1726 from accepting connect indications.

1727 If *fd* refers to a connectionless-mode service, only one endpoint may be associated with a
 1728 protocol address. If a user attempts to bind a second transport endpoint to an already bound
 1729 protocol address, *t_bind()* will return -1 and set *t_errno* to [TADDRBUSY].

1730 VALID STATES

1731 T_UNBND

1732 ERRORS

1733 On failure, *t_errno* is set to one of the following:

1734	[TBADF]	The specified file descriptor does not refer to a transport endpoint.
1735	[TOUTSTATE]	The function was issued in the wrong sequence.
1736	[TBADADDR]	The specified protocol address was in an incorrect format or contained 1737 illegal information.
1738	[TNOADDR]	The transport provider could not allocate an address.
1739	[TACCES]	The user does not have permission to use the specified address.
1740	[TBUFOVFLW]	The number of bytes allowed for an incoming argument (<i>maxlen</i>) is 1741 greater than 0 but not sufficient to store the value of that argument. The 1742 provider's state will change to T_IDLE and the information to be returned 1743 in <i>ret</i> will be discarded.
1744	[TSYSERR]	A system error has occurred during execution of this function.
1745	[TADDRBUSY]	The requested address is in use.

1746 [TPROTO] This error indicates that a communication problem has been detected
1747 between XTI and the transport provider for which there is no other
1748 suitable XTI (*t_errno*).

1749 **RETURN VALUE**

1750 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
1751 *t_errno* is set to indicate an error.

1752 **SEE ALSO**

1753 *t_alloc()*, *t_close()*, *t_open()*, *t_optmgmt()*, *t_unbind()*.

1754 **CAVEATS**

1755 The requirement that the value of *qlen* never be negotiated from a requested value greater than
1756 zero to zero implies that transport providers, rather than the XTI implementation itself, accept
1757 this restriction.

1758 A transport provider may not allow an explicit binding of more than one transport endpoint to
1759 the same protocol address, although it allows more than one connection to be accepted for the
1760 same protocol address. To ensure portability, it is, therefore, recommended not to bind transport
1761 endpoints that are used as responding endpoints (*resfd*) in a call to *t_accept()*, if the responding
1762 address is to be the same as the called address.

1763 **CHANGE HISTORY**

1764 **Issue 4**

1765 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

1766 **NAME**

1767 t_close - close a transport endpoint

1768 **SYNOPSIS**

1769 #include <xti.h>

1770 int t_close(int *fd*);1771 **DESCRIPTION**

1772

1773

1774

Parameters	Before call	After call
<i>fd</i>	x	/

1775 The *t_close()* function informs the transport provider that the user is finished with the transport
 1776 endpoint specified by *fd*, and frees any local library resources associated with the endpoint. In
 1777 addition, *t_close()* closes the file associated with the transport endpoint.

1778 The function *t_close()* should be called from the T_UNBND state (see *t_getstate()*). However,
 1779 this function does not check state information, so it may be called from any state to close a
 1780 transport endpoint. If this occurs, the local library resources associated with the endpoint will
 1781 be freed automatically. In addition, *close()* will be issued for that file descriptor; the *close()*
 1782 will be abortive if there are no other descriptors in this, or in another process which references the
 1783 transport endpoint, and in this case will break any transport connection that may be associated
 1784 with that endpoint.

1785 A *t_close()* issued on a connection endpoint may cause data previously sent, or data not yet
 1786 received, to be lost. It is the responsibility of the transport user to ensure that data is received by
 1787 the remote peer.

1788 **VALID STATES**

1789 ALL - apart from T_UNINIT

1790 **ERRORS**1791 On failure, *t_errno* is set to the following:

1792 [TBADF] The specified file descriptor does not refer to a transport endpoint.

1793 [TPROTO] This error indicates that a communication problem has been detected
 1794 between XTI and the transport provider for which there is no other
 1795 suitable XTI (*t_errno*).

1796 **RETURN VALUE**

1797 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
 1798 *t_errno* is set to indicate an error.

1799 **SEE ALSO**1800 *t_getstate()*, *t_open()*, *t_unbind()*.1801 **CHANGE HISTORY**1802 **Issue 4**1803 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

1804 **NAME**1805 `t_connect` - establish a connection with another transport user1806 **SYNOPSIS**1807 `#include <xti.h>`1808 `int t_connect(int fd, struct t_call *sndcall, struct t_call *rcvcall);`1809 **DESCRIPTION**

1810

1811

1812

1813

1814

1815

1816

1817

1818

1819

1820

1821

1822

1823

1824

1825

1826

1827

1828

1829

1830

1831

1832

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

1833

1834

1835

1836

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

1837

1838

1839

1840

```

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

```

1841

1842

1843

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

1844

1845

1846

1847

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.

1848

1849

1850

1851

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

1852 The *opt* argument permits users to define the options that may be passed to the transport
 1853 provider. These options are specific to the underlying protocol of the transport provider and are
 1854 described for ISO and TCP protocols in Appendix A on page 189, Appendix B on page 199 and
 1855 Appendix F on page 253. The user may choose not to negotiate protocol options by setting the
 1856 *len* field of *opt* to zero. In this case, the provider may use default options.

1857 If used, *sndcall->opt.buf* must point to a buffer with the corresponding options; the *maxlen* and *buf*
 1858 fields of the **netbuf** structure pointed by *rcvcall->addr* and *rcvcall->opt* must be set before the call.

1859 The *udata* argument enables the caller to pass user data to the destination transport user and
 1860 receive user data from the destination user during connection establishment. However, the
 1861 amount of user data must not exceed the limits supported by the transport provider as returned
 1862 in the *connect* field of the *info* argument of *t_open()* or *t_getinfo()*. If the *len* of *udata* is zero in
 1863 *sndcall*, no data will be sent to the destination transport user.

1864 On return, the *addr*, *opt* and *udata* fields of *rcvcall* will be updated to reflect values associated
 1865 with the connection. Thus, the *maxlen* field of each argument must be set before issuing this
 1866 function to indicate the maximum size of the buffer for each. However, *rcvcall* may be a null
 1867 pointer, in which case no information is given to the user on return from *t_connect()*.

1868 By default, *t_connect()* executes in synchronous mode, and will wait for the destination user's
 1869 response before returning control to the local user. A successful return (that is, return value of
 1870 zero) indicates that the requested connection has been established. However, if **O_NONBLOCK**
 1871 is set (via *t_open()* or *fcntl()*), *t_connect()* executes in asynchronous mode. In this case, the call
 1872 will not wait for the remote user's response, but will return control immediately to the local user
 1873 and return -1 with *t_errno* set to **[TNODATA]** to indicate that the connection has not yet been
 1874 established. In this way, the function simply initiates the connection establishment procedure
 1875 by sending a connect request to the destination transport user. The *t_rcvconnect()* function is
 1876 used in conjunction with *t_connect()* to determine the status of the requested connection.

1877 When a synchronous *t_connect()* call is interrupted by the arrival of a signal, the state of the
 1878 corresponding transport endpoint is **T_OUTCON**, allowing a further call to either *t_rcvconnect()*,
 1879 *t_rcvdis()* or *t_snddis()*.

1880 VALID STATES

1881 **T_IDLE**

1882 ERRORS

1883 On failure, *t_errno* is set to one of the following:

1884 **[TBADF]** The specified file descriptor does not refer to a transport endpoint.

1885 **[TOUTSTATE]** The function was issued in the wrong sequence.

1886 **[TNODATA]** **O_NONBLOCK** was set, so the function successfully initiated the
 1887 connection establishment procedure, but did not wait for a response from
 1888 the remote user.

1889 **[TBADADDR]** The specified protocol address was in an incorrect format or contained
 1890 illegal information.

1891 **[TBADOPT]** The specified protocol options were in an incorrect format or contained
 1892 illegal information.

1893 **[TBADDATA]** The amount of user data specified was not within the bounds allowed by
 1894 the transport provider.

1895 **[TACCES]** The user does not have permission to use the specified address or
 1896 options.

1897	[TBUFOVFLW]	The number of bytes allocated for an incoming argument (<i>maxlen</i>) is greater than 0 but not sufficient to store the value of that argument. If
1898		executed in synchronous mode, the provider's state, as seen by the user,
1899		changes to T_DATAXFER, and the information to be returned in <i>rcvcall</i> is
1900		discarded.
1901		
1902	[TLOOK]	An asynchronous event has occurred on this transport endpoint and
1903		requires immediate attention.
1904	[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
1905	[TSYSERR]	A system error has occurred during execution of this function.
1906	[TADDRBUSY]	This transport provider does not support multiple connections with the
1907		same local and remote addresses. This error indicates that a connection
1908		already exists.
1909	[TPROTO]	This error indicates that a communication problem has been detected
1910		between XTI and the transport provider for which there is no other
1911		suitable XTI (<i>t_errno</i>).
1912	RETURN VALUE	
1913		Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
1914		<i>t_errno</i> is set to indicate an error.
1915	SEE ALSO	
1916		<i>t_accept()</i> , <i>t_alloc()</i> , <i>t_getinfo()</i> , <i>t_listen()</i> , <i>t_open()</i> , <i>t_optmgmt()</i> , <i>t_rcvconnect()</i> .
1917	CHANGE HISTORY	
1918	Issue 4	
1919		The SYNOPSIS section is placed in the form of a standard C function prototype.

1920 **NAME**

1921 t_error - produce error message

1922 **SYNOPSIS**

1923 #include <xti.h>

1924 int t_error(char *errmsg);

1925 **DESCRIPTION**

1926

1927

1928

Parameters	Before call	After call
errmsg	x	/

1929 The *t_error()* function produces a language-dependent message on the standard error output
 1930 which describes the last error encountered during a call to a transport function. The argument
 1931 string *errmsg* is a user-supplied error message that gives context to the error.

1932 The error message is written as follows: first (if *errmsg* is not a null pointer and the character
 1933 pointed to by *errmsg* is not the null character) the string pointed to by *errmsg* followed by a colon
 1934 and a space; then a standard error message string for the current error defined in *t_errno*. If
 1935 *t_errno* has a value different from [TSYSERR], the standard error message string is followed by a
 1936 newline character. If, however, *t_errno* is equal to [TSYSERR], the *t_errno* string is followed by
 1937 the standard error message string for the current error defined in *errno* followed by a newline.

1938 The language for error message strings written by *t_error()* is implementation-defined. If it is in
 1939 English, the error message string describing the value in *t_errno* is identical to the comments
 1940 following the *t_errno* codes defined in *xti.h*. The contents of the error message strings describing
 1941 the value in *errno* are the same as those returned by the *strerror(3C)* function with an argument
 1942 of *errno*.

1943 The error number, *t_errno*, is only set when an error occurs and it is not cleared on successful
 1944 calls.

1945 **EXAMPLE**

1946 If a *t_connect()* function fails on transport endpoint *fd2* because a bad address was given, the
 1947 following call might follow the failure:

```
1948     t_error("t_connect failed on fd2");
```

1949 The diagnostic message to be printed would look like:

```
1950     t_connect failed on fd2: incorrect addr format
```

1951 where *incorrect addr format* identifies the specific error that occurred, and *t_connect failed on fd2*
 1952 tells the user which function failed on which transport endpoint.

1953 **VALID STATES**

1954 All - apart from T_UNINIT

1955 **ERRORS**

1956 No errors are defined for the *t_error()* function.

1957 **RETURN VALUE**

1958 Upon completion, a value of 0 is returned.

1959 **CHANGE HISTORY**1960 **Issue 4**

1961 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

1962 **NAME**

1963 t_free - free a library structure

1964 **SYNOPSIS**

1965 #include <xti.h>

1966 int t_free(char *ptr, int struct_type);

1967 **DESCRIPTION**

1968

1969

Parameters	Before call	After call
<i>ptr</i>	x	/
<i>struct_type</i>	x	/

1970

1971

1972 The *t_free()* function frees memory previously allocated by *t_alloc()*. This function will free
 1973 memory for the specified structure, and will also free memory for buffers referenced by the
 1974 structure.

1975 The argument *ptr* points to one of the seven structure types described for *t_alloc()*, and
 1976 *struct_type* identifies the type of that structure which must be one of the following:

1977	T_BIND	struct	t_bind
1978	T_CALL	struct	t_call
1979	T_OPTMGMT	struct	t_optmgmt
1980	T_DIS	struct	t_discon
1981	T_UNITDATA	struct	t_unitdata
1982	T_UDERROR	struct	t_uderr
1983	T_INFO	struct	t_info

1984 where each of these structures is used as an argument to one or more transport functions.

1985 The function *t_free()* will check the *addr*, *opt* and *udata* fields of the given structure (as
 1986 appropriate) and free the buffers pointed to by the *buf* field of the **netbuf** structure. If *buf* is a
 1987 null pointer, *t_free()* will not attempt to free memory. After all buffers are freed, *t_free()* will free
 1988 the memory associated with the structure pointed to by *ptr*.

1989 Undefined results will occur if *ptr* or any of the *buf* pointers points to a block of memory that
 1990 was not previously allocated by *t_alloc()*.

1991 **VALID STATES**

1992 ALL - apart from T_UNINIT

1993 **ERRORS**1994 On failure, *t_errno* is set to the following:

1995 [TSYSERR] A system error has occurred during execution of this function.

1996 [TNOSTRUCTYPE] Unsupported *struct_type* requested.

1997 [TPROTO] This error indicates that a communication problem has been detected
 1998 between XTI and the transport provider for which there is no other
 1999 suitable XTI (*t_errno*).

2000 **RETURN VALUE**

2001 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
 2002 *t_errno* is set to indicate an error.

2003 **SEE ALSO**2004 *t_alloc()*.

2005 **CHANGE HISTORY**

2006 **Issue 4**

2007 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2008 **NAME**2009 `t_getinfo` - get protocol-specific service information2010 **SYNOPSIS**2011 `#include <xti.h>`2012 `int t_getinfo(int fd, struct t_info *info);`2013 **DESCRIPTION**2014
2015

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

2016
2017
2018
2019
2020
2021
2022
2023
2024

2025 This function returns the current characteristics of the underlying transport protocol and/or
2026 transport connection associated with file descriptor *fd*. The *info* pointer is used to return the
2027 same information returned by *t_open()*, although not necessarily precisely the same values. This
2028 function enables a transport user to access this information during any phase of communication.

2029 This argument points to a **t_info** structure which contains the following members:

```

2030     long addr;      /* max size of the transport protocol address      */
2031     long options;  /* max number of bytes of protocol-specific options */
2032     long tsdu;    /* max size of a transport service data unit (TSDU) */
2033     long etsdu;   /* max size of an expedited transport service      */
2034                  /* data unit (ETSDU)                               */
2035     long connect; /* max amount of data allowed on connection        */
2036                  /* establishment functions                          */
2037     long discon;  /* max amount of data allowed on t_snddis()        */
2038                  /* and t_rcvdis() functions                        */
2039     long servtype; /* service type supported by the transport provider */
2040     long flags;   /* other info about the transport provider          */

```

2041 The values of the fields have the following meanings:

2042 **addr** A value greater than zero indicates the maximum size of a transport
2043 protocol address and a value of -2 specifies that the transport provider
2044 does not provide user access to transport protocol addresses.

2045 **options** A value greater than zero indicates the maximum number of bytes of
2046 protocol-specific options supported by the provider, and a value of -2
2047 specifies that the transport provider does not support user-settable
2048 options.

2049 **tsdu** A value greater than zero specifies the maximum size of a transport
2050 service data unit (TSDU); a value of zero specifies that the transport
2051 provider does not support the concept of TSDU, although it does support
2052 the sending of a datastream with no logical boundaries preserved across a
2053 connection; a value of -1 specifies that there is no limit on the size of a
2054 TSDU; and a value of -2 specifies that the transfer of normal data is not
2055 supported by the transport provider.

2056	<i>etsdu</i>	A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider. Note that the semantics of expedited data may be quite different for different transport providers (see Appendix A on page 189 and Appendix B on page 199).
2066	<i>connect</i>	A value greater than zero specifies the maximum amount of data that may be associated with connection establishment functions and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions.
2070	<i>discon</i>	A value greater than zero specifies the maximum amount of data that may be associated with the <i>t_snddis()</i> and <i>t_rcvdis()</i> functions and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions.
2074	<i>servtype</i>	This field specifies the service type supported by the transport provider, as described below.
2076	<i>flags</i>	This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates that the underlying transport provider supports the sending of zero-length TSDUs. See Appendix A on page 189 for a discussion of the separate issue of zero-length fragments within a TSDU.
2081		If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the <i>t_alloc()</i> function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of protocol option negotiation during connection establishment (the <i>t_optmgmt()</i> call has no affect on the values returned by <i>t_getinfo()</i>). These values will only change from the values presented to <i>t_open()</i> after the endpoint enters the T_DATAXFER state.
2088		The <i>servtype</i> field of <i>info</i> specifies one of the following values on return:
2089	T_COTS	The transport provider supports a connection-mode service but does not support the optional orderly release facility.
2091	T_COTS_ORD	The transport provider supports a connection-mode service with the optional orderly release facility.
2093	T_CLTS	The transport provider supports a connectionless-mode service. For this service type, <i>t_open()</i> will return -2 for <i>etsdu</i> , <i>connect</i> and <i>discon</i> .
2095	VALID STATES	
2096		ALL - apart from T_UNINIT
2097	ERRORS	
2098		On failure, <i>t_errno</i> is set to one of the following:
2099	[TBADF]	The specified file descriptor does not refer to a transport endpoint.
2100	[TSYSERR]	A system error has occurred during execution of this function.

2101 [TPROTO] This error indicates that a communication problem has been detected
2102 between XTI and the transport provider for which there is no other
2103 suitable XTI (*t_errno*).

2104 **RETURN VALUE**

2105 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
2106 *t_errno* is set to indicate an error.

2107 **SEE ALSO**

2108 *t_alloc()*, *t_open()*.

2109 **CHANGE HISTORY**

2110 **Issue 4**

2111 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2112 **NAME**

2113 t_getprotaddr - get the protocol addresses

2114 **SYNOPSIS**

2115 #include <xti.h>

2116 int t_getprotaddr(int fd, struct t_bind *boundaddr, struct t_bind *peeraddr);

2117 **DESCRIPTION**

2118
2119

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

2120
2121
2122
2123
2124
2125
2126
2127
2128

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

2138 **VALID STATES**

2139 ALL - apart from T_UNINIT

2140 **ERRORS**

2141 On failure, *t_errno* is set to one of the following:

2142 [TBADF]	The specified file descriptor does not refer to a transport endpoint.
2143 [TBUFOVFLW]	The number of bytes allocated for an incoming argument (<i>maxlen</i>) is greater than 0 but not sufficient to store the value of that argument.
2144	
2145 [TSYSERR]	A system error has occurred during execution of this function.
2146 [TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (<i>t_errno</i>).
2147	
2148	

2149 **RETURN VALUE**

2150 Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned
2151 and *t_errno* is set to indicate the error.

2152 **SEE ALSO**

2153 *t_bind()*.

2154 **CHANGE HISTORY**

2155 **Issue 4**

2156 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2157 **NAME**

2158 t_getstate - get the current state

2159 **SYNOPSIS**

2160 #include <xti.h>

2161 int t_getstate(int *fd*);

2162 **DESCRIPTION**

2163

2164

2165

Parameters	Before call	After call
<i>fd</i>	x	/

2166 The *t_getstate()* function returns the current state of the provider associated with the transport
2167 endpoint specified by *fd*.

2168 **VALID STATES**

2169 ALL - apart from T_UNINIT

2170 **ERRORS**

2171 On failure, *t_errno* is set to one of the following:

2172 [TBADF] The specified file descriptor does not refer to a transport endpoint.

2173 [TSTATECHNG] The transport provider is undergoing a transient state change.

2174 [TSYSERR] A system error has occurred during execution of this function.

2175 [TPROTO] This error indicates that a communication problem has been detected
2176 between XTI and the transport provider for which there is no other
2177 suitable XTI (*t_errno*).

2178 **RETURN VALUE**

2179 State is returned upon successful completion. Otherwise, a value of -1 is returned and *t_errno* is
2180 set to indicate an error. The current state is one of the following:

2181 T_UNBND Unbound.

2182 T_IDLE Idle.

2183 T_OUTCON Outgoing connection pending.

2184 T_INCON Incoming connection pending.

2185 T_DATAXFER Data transfer.

2186 T_OUTREL Outgoing orderly release (waiting for an orderly release indication).

2187 T_INREL Incoming orderly release (waiting to send an orderly release request).

2188 If the provider is undergoing a state transition when *t_getstate()* is called, the function will fail.

2189 **SEE ALSO**

2190 *t_open()*.

2191 **CHANGE HISTORY**

2192 **Issue 4**

2193 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2194 **NAME**

2195 t_listen - listen for a connect indication

2196 **SYNOPSIS**

2197 #include <xti.h>

2198 int t_listen(int *fd*, struct t_call **call*);2199 **DESCRIPTION**2200
2201

	Parameters	Before call	After call
2202	<i>fd</i>	x	/
2203	<i>call</i> -> <i>addr.maxlen</i>	x	/
2204	<i>call</i> -> <i>addr.len</i>	/	x
2205	<i>call</i> -> <i>addr.buf</i>	?	(?)
2206	<i>call</i> -> <i>opt.maxlen</i>	x	/
2207	<i>call</i> -> <i>opt.len</i>	/	x
2208	<i>call</i> -> <i>opt.buf</i>	?	(?)
2209	<i>call</i> -> <i>udata.maxlen</i>	x	/
2210	<i>call</i> -> <i>udata.len</i>	/	x
2211	<i>call</i> -> <i>udata.buf</i>	?	(?)
2212	<i>call</i> -> <i>sequence</i>	/	x

2213 This function listens for a connect request from a calling transport user. The argument *fd*
 2214 identifies the local transport endpoint where connect indications arrive, and on return, *call*
 2215 contains information describing the connect indication. The parameter *call* points to a **t_call**
 2216 structure which contains the following members:

```
2217     struct netbuf addr;
2218     struct netbuf opt;
2219     struct netbuf udata;
2220     int sequence;
```

2221 In *call*, *addr* returns the protocol address of the calling transport user. This address is in a format
 2222 usable in future calls to *t_connect()*. Note, however that *t_connect()* may fail for other reasons,
 2223 for example [TADDRBUSY]. *opt* returns options associated with the connect request, *udata*
 2224 returns any user data sent by the caller on the connect request, and *sequence* is a number that
 2225 uniquely identifies the returned connect indication. The value of *sequence* enables the user to
 2226 listen for multiple connect indications before responding to any of them.

2227 Since this function returns values for the *addr*, *opt* and *udata* fields of *call*, the *maxlen* field of each
 2228 must be set before issuing the *t_listen()* to indicate the maximum size of the buffer for each.

2229 By default, *t_listen()* executes in synchronous mode and waits for a connect indication to arrive
 2230 before returning to the user. However, if O_NONBLOCK is set via *t_open()* or *fcntl()*, *t_listen()*
 2231 executes asynchronously, reducing to a poll for existing connect indications. If none are
 2232 available, it returns -1 and sets *t_errno* to [TNODATA].

2233 **VALID STATES**

2234 T_IDLE, T_INCON

2235 **ERRORS**2236 On failure, *t_errno* is set to one of the following:

2237 [TBADF] The specified file descriptor does not refer to a transport endpoint.
 2238 [TBADQLEN] The argument *qlen* of the endpoint referenced by *fd* is zero.

2239	[TBUFOVFLW]	The number of bytes allocated for an incoming argument (<i>maxlen</i>) is greater than 0 but not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect indication information to be returned in <i>call</i> is discarded. The value of <i>sequence</i> returned can be used to do a <i>t_snddis()</i> .
2240		
2241		
2242		
2243		
2244	[TNODATA]	O_NONBLOCK was set, but no connect indications had been queued.
2245	[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
2246		
2247	[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
2248	[TOUTSTATE]	The function was issued in the wrong sequence on the transport endpoint referenced by <i>fd</i> .
2249		
2250	[TSYSERR]	A system error has occurred during execution of this function.
2251	[TQFULL]	The maximum number of outstanding indications has been reached for the endpoint referenced by <i>fd</i> .
2252		
2253	[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (<i>t_errno</i>).
2254		
2255		
2256	CAVEATS	
2257		Some transport providers do not differentiate between a connect indication and the connection itself. If this is the case, a successful return of <i>t_listen()</i> indicates an existing connection (see Appendix B on page 199).
2258		
2259		
2260	RETURN VALUE	
2261		Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and <i>t_errno</i> is set to indicate an error.
2262		
2263	SEE ALSO	
2264		<i>fcntl()</i> , <i>t_accept()</i> , <i>t_alloc()</i> , <i>t_bind()</i> , <i>t_connect()</i> , <i>t_open()</i> , <i>t_optmgmt()</i> , <i>t_rcvconnect()</i> .
2265	CHANGE HISTORY	
2266	Issue 4	
2267		The SYNOPSIS section is placed in the form of a standard C function prototype.

2268 **NAME**

2269 t_look - look at the current event on a transport endpoint

2270 **SYNOPSIS**

```
2271 #include <xti.h>
2272 int t_look(int fd);
```

2273 **DESCRIPTION**

2274
2275
2276

Parameters	Before call	After call
<i>fd</i>	x	/

2277 This function returns the current event on the transport endpoint specified by *fd*. This function
2278 enables a transport provider to notify a transport user of an asynchronous event when the user
2279 is calling functions in synchronous mode. Certain events require immediate notification of the
2280 user and are indicated by a specific error, [TLOOK], on the current or next function to be
2281 executed. Details on events which cause functions to fail [TLOOK] may be found in Section 5.6
2282 on page 34.

2283 This function also enables a transport user to poll a transport endpoint periodically for
2284 asynchronous events.

2285 **VALID STATES**

2286 ALL - apart from T_UNINIT

2287 **ERRORS**2288 On failure, *t_errno* is set to one of the following:

2289 [TBADF] The specified file descriptor does not refer to a transport endpoint.
2290 [TSYSERR] A system error has occurred during execution of this function.
2291 [TPROTO] This error indicates that a communication problem has been detected
2292 between XTI and the transport provider for which there is no other
2293 suitable XTI (*t_errno*).

2294 **RETURN VALUE**

2295 Upon success, *t_look()* returns a value that indicates which of the allowable events has occurred,
2296 or returns zero if no event exists. One of the following events is returned:

2297 T_LISTEN Connection indication received.
2298 T_CONNECT Connect confirmation received.
2299 T_DATA Normal data received.
2300 T_EXDATA Expedited data received.
2301 T_DISCONNECT Disconnect received.
2302 T_UDERR Datagram error indication.
2303 T_ORDREL Orderly release indication.
2304 T_GODATA Flow control restrictions on normal data flow that led to a [TFLOW] error
2305 have been lifted. Normal data may be sent again.
2306 T_GOEXDATA Flow control restrictions on expedited data flow that led to a [TFLOW]
2307 error have been lifted. Expedited data may be sent again.

2308 On failure, -1 is returned and *t_errno* is set to indicate the error.

2309 **SEE ALSO**2310 *t_open()*, *t_snd()*, *t_sndudata()*.2311 **APPLICATION USAGE**

2312 Additional functionality is provided through the Event Management (EM) interface.

2313 **CHANGE HISTORY**2314 **Issue 4**2315 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2316 NAME

2317 t_open - establish a transport endpoint

2318 SYNOPSIS

2319 #include <xti.h>

2320 #include <fcntl.h>

2321 int t_open(char *name, int oflag, struct t_info *info);

2322 DESCRIPTION

2323

2324

2325

2326

2327

2328

2329

2330

2331

2332

2333

2334

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

2335 The *t_open()* function must be called as the first step in the initialisation of a transport endpoint.
 2336 This function establishes a transport endpoint by supplying a transport provider identifier that
 2337 indicates a particular transport provider (that is, transport protocol) and returning a file
 2338 descriptor that identifies that endpoint.

2339 The argument *name* points to a transport provider identifier and *oflag* identifies any open flags
 2340 (as in *open()*). The argument *oflag* is constructed from O_RDWR optionally bitwise inclusive-
 2341 OR'ed with O_NONBLOCK. These flags are defined by the header <fcntl.h>. The file
 2342 descriptor returned by *t_open()* will be used by all subsequent functions to identify the
 2343 particular local transport endpoint.

2344 This function also returns various default characteristics of the underlying transport protocol by
 2345 setting fields in the **t_info** structure. This argument points to a **t_info** which contains the
 2346 following members:

```

2347     long addr;          /* max size of the transport protocol address */
2348     long options;      /* max number of bytes of */
2349                       /* protocol-specific options */
2350     long tsdu;         /* max size of a transport service data */
2351                       /* unit (TSDU) */
2352     long etsdu;        /* max size of an expedited transport */
2353                       /* service data unit (ETSDU) */
2354     long connect;      /* max amount of data allowed on */
2355                       /* connection establishment functions */
2356     long discon;       /* max amount of data allowed on */
2357                       /* t_snddis() and t_rcvdis() functions */
2358     long servtype;     /* service type supported by the */
2359                       /* transport provider */
2360     long flags;        /* other info about the transport provider */

```

2361		The values of the fields have the following meanings:
2362	<i>addr</i>	A value greater than zero indicates the maximum size of a transport protocol address and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses.
2363		
2364		
2365	<i>options</i>	A value greater than zero indicates the maximum number of bytes of protocol-specific options supported by the provider and a value of -2 specifies that the transport provider does not support user-settable options.
2366		
2367		
2368		
2369	<i>tsdu</i>	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 to the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider.
2370		
2371		
2372		
2373		
2374		
2375		
2376	<i>etsdu</i>	A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider. Note that the semantics of expedited data may be quite different for different transport providers (see Appendix A on page 189 and Appendix B on page 199).
2377		
2378		
2379		
2380		
2381		
2382		
2383		
2384		
2385		
2386	<i>connect</i>	A value greater than zero specifies the maximum amount of data that may be associated with connection establishment functions and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions.
2387		
2388		
2389		
2390	<i>discon</i>	A value greater than zero specifies the maximum amount of data that may be associated with the <i>t_snddis()</i> and <i>t_rcvdis()</i> functions and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions.
2391		
2392		
2393		
2394	<i>servtype</i>	This field specifies the service type supported by the transport provider, as described below.
2395		
2396	<i>flags</i>	This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates the underlying transport provider supports the sending of zero-length TSDUs. See Appendix A on page 189 for a discussion of the separate issue of zero-length fragments within a TSDU.
2397		
2398		
2399		
2400		
2401		If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the <i>t_alloc()</i> function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.
2402		
2403		
2404		
2405		The <i>servtype</i> field of <i>info</i> specifies one of the following values on return:
2406	T_COTS	The transport provider supports a connection-mode service but does not support the optional orderly release facility.
2407		

- 2408 T_COTS_ORD The transport provider supports a connection-mode service with the
2409 optional orderly release facility.
- 2410 T_CLTS The transport provider supports a connectionless-mode service. For this
2411 service type, *t_open()* will return -2 for *etsdu*, *connect* and *discon*.
- 2412 A single transport endpoint may support only one of the above services at one time.
- 2413 If *info* is set to a null pointer by the transport user, no protocol information is returned by
2414 *t_open()*.
- 2415 **VALID STATES**
- 2416 T_UNINIT
- 2417 **ERRORS**
- 2418 On failure, *t_errno* is set to the following:
- 2419 [TBADFLAG] An invalid flag is specified.
- 2420 [TBADNAME] Invalid transport provider name.
- 2421 [TSYSERR] A system error has occurred during execution of this function.
- 2422 [TPROTO] This error indicates that a communication problem has been detected
2423 between XTI and the transport provider for which there is no other
2424 suitable XTI (*t_errno*).
- 2425 **RETURN VALUES**
- 2426 A valid file descriptor is returned upon successful completion. Otherwise, a value of -1 is
2427 returned and *t_errno* is set to indicate an error.
- 2428 **SEE ALSO**
- 2429 *open()*.
- 2430 **CHANGE HISTORY**
- 2431 **Issue 4**
- 2432 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2433 NAME

2434 t_optmgmt - manage options for a transport endpoint

2435 SYNOPSIS

2436 #include <xti.h>

2437 int t_optmgmt(int *fd*, struct t_optmgmt **req*, struct t_optmgmt **ret*);

2438 DESCRIPTION

2439

2440

2441

2442

2443

2444

2445

2446

2447

2448

2449

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

2450 The *t_optmgmt()* function enables a transport user to retrieve, verify or negotiate protocol
 2451 options with the transport provider. The argument *fd* identifies a transport endpoint.

2452 The *req* and *ret* arguments point to a **t_optmgmt** structure containing the following members:

2453 struct netbuf opt;

2454 long flags;

2455 The *opt* field identifies protocol options and the *flags* field is used to specify the action to take
 2456 with those options.

2457 The options are represented by a **netbuf** structure in a manner similar to the address in *t_bind()*.
 2458 The argument *req* is used to request a specific action of the provider and to send options to the
 2459 provider. The argument *len* specifies the number of bytes in the options, *buf* points to the
 2460 options buffer, and *maxlen* has no meaning for the *req* argument. The transport provider may
 2461 return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum
 2462 size of the options buffer and *buf* points to the buffer where the options are to be placed. On
 2463 return, *len* specifies the number of bytes of options returned. The value in *maxlen* has no
 2464 meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum
 2465 number of bytes the options buffer can hold.

2466 Each option in the options buffer is of the form **struct t_opthdr** possibly followed by an option
 2467 value.

2468 The *level* field of **struct t_opthdr** identifies the XTI level or a protocol of the transport provider.
 2469 The *name* field identifies the option within the level, and *len* contains its total length; that is, the
 2470 length of the option header **t_opthdr** plus the length of the option value. If *t_optmgmt()* is called
 2471 with the action T_NEGOTIATE set, the *status* field of the returned options contains information
 2472 about the success or failure of a negotiation.

2473 Each option in the input or output option buffer must start at a long-word boundary. The macro
 2474 **OPT_NEXTHDR(pbuf, buflen, poption)** can be used for that purpose. The parameter *pbuf*
 2475 denotes a pointer to an option buffer *opt.buf*, and *buflen* is its length. The parameter *poption*
 2476 points to the current option in the option buffer. **OPT_NEXTHDR** returns a pointer to the
 2477 position of the next option or returns a null pointer if the option buffer is exhausted. The macro
 2478 is helpful for writing and reading. See <xti.h> in Appendix F on page 253 for the exact
 2479 definition.

2480 If the transport user specifies several options on input, all options must address the same level.

2481 If any option in the options buffer does not indicate the same level as the first option, or the level
 2482 specified is unsupported, then the `t_optmgmt()` request will fail with [TBADOPT]. If the error is
 2483 detected, some options have possibly been successfully negotiated. The transport user can
 2484 check the current status by calling `t_optmgmt()` with the T_CURRENT flag set.

2485 Chapter 6 contains a detailed description about the use of options and should be read before
 2486 using this function.

2487 The *flags* field of *req* must specify one of the following actions:

2488 T_NEGOTIATE This action enables the transport user to negotiate option values.

2489 The user specifies the options of interest and their values in the buffer
 2490 specified by *req->opt.buf* and *req->opt.len*. The negotiated option values
 2491 are returned in the buffer pointed to by *ret->opt.buf*. The *status* field of
 2492 each returned option is set to indicate the result of the negotiation. The
 2493 value is T_SUCCESS if the proposed value was negotiated, T_PARTSUCCESS if a degraded value was negotiated, T_FAILURE if the
 2494 negotiation failed (according to the negotiation rules), T_NOTSUPPORT
 2495 if the transport provider does not support this option or illegally requests
 2496 negotiation of a privileged option, and T_READONLY if modification of a
 2497 read-only option was requested. If the status is T_SUCCESS,
 2498 T_FAILURE, T_NOTSUPPORT or T_READONLY, the returned option
 2499 value is the same as the one requested on input.
 2500

2501 The overall result of the negotiation is returned in *ret->flags*.

2502 This field contains the worst single result, whereby the rating is done
 2503 according to the order T_NOTSUPPORT, T_READONLY, T_FAILURE,
 2504 T_PARTSUCCESS, T_SUCCESS. The value T_NOTSUPPORT is the
 2505 worst result and T_SUCCESS is the best.

2506 For each level, the option T_ALLOPT (see below) can be requested on
 2507 input. No value is given with this option; only the **t_opthdr** part is
 2508 specified. This input requests to negotiate all supported options of this
 2509 level to their default values. The result is returned option by option in
 2510 *ret->opt.buf*. (Note that depending on the state of the transport endpoint,
 2511 not all requests to negotiate the default value may be successful.)

2512 T_CHECK This action enables the user to verify whether the options specified in *req*
 2513 are supported by the transport provider.

2514 If an option is specified with no option value (it consists only of a
 2515 **t_opthdr** structure), the option is returned with its *status* field set to
 2516 T_SUCCESS if it is supported, T_NOTSUPPORT if it is not or needs
 2517 additional user privileges, and T_READONLY if it is read-only (in the
 2518 current XTI state). No option value is returned.

2519 If an option is specified with an option value, the *status* field of the
 2520 returned option has the same value, as if the user had tried to negotiate
 2521 this value with T_NEGOTIATE. If the status is T_SUCCESS, T_FAILURE,
 2522 T_NOTSUPPORT or T_READONLY, the returned option value is the
 2523 same as the one requested on input.

2524 The overall result of the option checks is returned in *ret->flags*. This field
 2525 contains the worst single result of the option checks, whereby the rating

2526 is the same as for T_NEGOTIATE.

2527 Note that no negotiation takes place. All currently effective option values
2528 remain unchanged.

2529 T_DEFAULT This action enables the transport user to retrieve the default option
2530 values. The user specifies the options of interest in *req->opt.buf*. The
2531 option values are irrelevant and will be ignored; it is sufficient to specify
2532 the **t_opthdr** part of an option only. The default values are then returned
2533 in *ret->opt.buf*.

2534 The *status* field returned is T_NOTSUPPORT if the protocol level does
2535 not support this option or the transport user illegally requested a
2536 privileged option, T_READONLY if the option is read-only, and set to
2537 T_SUCCESS in all other cases. The overall result of the request is
2538 returned in *ret->flags*. This field contains the worst single result, whereby
2539 the rating is the same as for T_NEGOTIATE.

2540 For each level, the option T_ALLOPT (see below) can be requested on
2541 input. All supported options of this level with their default values are
2542 then returned. In this case, *ret->opt.maxlen* must be given at least the
2543 value *info->options* (see *t_getinfo()*, *t_open()*) before the call.

2544 T_CURRENT This action enables the transport user to retrieve the currently effective
2545 option values. The user specifies the options of interest in *req->opt.buf*.
2546 The option values are irrelevant and will be ignored; it is sufficient to
2547 specify the **t_opthdr** part of an option only. The currently effective values
2548 are then returned in *ret->opt.buf*.

2549 The *status* field returned is T_NOTSUPPORT if the protocol level does
2550 not support this option or the transport user illegally requested a
2551 privileged option, T_READONLY if the option is read-only, and set to
2552 T_SUCCESS in all other cases. The overall result of the request is
2553 returned in *ret->flags*. This field contains the worst single result, whereby
2554 the rating is the same as for T_NEGOTIATE.

2555 For each level, the option T_ALLOPT (see below) can be requested on
2556 input. All supported options of this level with their currently effective
2557 values are then returned.

2558 The option T_ALLOPT can only be used with *t_optmgmt()* and the actions T_NEGOTIATE,
2559 T_DEFAULT and T_CURRENT. It can be used with any supported level and addresses all
2560 supported options of this level. The option has no value; it consists of a **t_opthdr** only. Since in
2561 a *t_optmgmt()* call only options of one level may be addressed, this option should not be
2562 requested together with other options. The function returns as soon as this option has been
2563 processed.

2564 Options are independently processed in the order they appear in the input option buffer. If an
2565 option is multiply input, it depends on the implementation whether it is multiply output or
2566 whether it is returned only once.

2567 Transport providers may not be able to provide an interface capable of supporting
2568 T_NEGOTIATE and/or T_CHECK functionalities. When this is the case, the error
2569 [TNOTSUPPORT] is returned.

2570 The function *t_optmgmt()* may block under various circumstances and depending on the
2571 implementation. The function will block, for instance, if the protocol addressed by the call
2572 resides on a separate controller. It may also block due to flow control constraints; that is, if data

2573 sent previously across this transport endpoint has not yet been fully processed. If the function is
 2574 interrupted by a signal, the option negotiations that have been done so far may remain valid.
 2575 The behaviour of the function is not changed if O_NONBLOCK is set.

2576 XTI-LEVEL OPTIONS

2577 XTI-level options are not specific for a particular transport provider. An XTI implementation
 2578 supports none, all or any subset of the options defined below. An implementation may restrict
 2579 the use of any of these options by offering them only in the privileged or read-only mode, or if *fd*
 2580 relates to specific transport providers.

2581 The subsequent options are not association-related (see **Chapter 5, The Use of Options**). They
 2582 may be negotiated in all XTI states except T_UNINIT.

2583 The protocol level is XTI_GENERIC. For this level, the following options are defined:

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

2594 **Table 7-1** XTI-level Options

2595 A request for XTI_DEBUG is an absolute requirement. A request to activate XTI_LINGER is an
 2596 absolute requirement; the timeout value to this option is not. XTI_RCVBUF, XTI_RCVLOWAT,
 2597 XTI_SNDBUF and XTI_SNDLOWAT are not absolute requirements.

2598 XTI_DEBUG This option enables debugging. The values of this option are
 2599 implementation-defined. Debugging is disabled if the option is specified
 2600 with “no value”; that is, with an option header only.

2601 The system supplies utilities to process the traces. Note that an
 2602 implementation may also provide other means for debugging.

2603 XTI_LINGER This option is used to linger the execution of a *t_close()* or *close()* if send
 2604 data is still queued in the send buffer. The option value specifies the
 2605 linger period. If a *close()* or *t_close()* is issued and the send buffer is not
 2606 empty, the system attempts to send the pending data within the linger
 2607 period before closing the endpoint. Data still pending after the linger
 2608 period has elapsed is discarded.

2609 Depending on the implementation, *t_close()* or *close()* either block for at
 2610 maximum the linger period, or immediately return, whereupon the
 2611 system holds the connection in existence for at most the linger period.

2612 The option value consists of a structure **t_linger** declared as:

```
2613 struct t_linger {
2614     long l_onoff; /* switch option on/off */
2615     long l_linger; /* linger period in seconds */
2616 }
```

2617		Legal values for the field <i>l_onoff</i> are:
2618		T_NO switch option off
2619		T_YES activate option
2620		The value <i>l_onoff</i> is an absolute requirement.
2621		The field <i>l_linger</i> determines the linger period in seconds. The transport
2622		user can request the default value by setting the field to T_UNSPEC. The
2623		default timeout value depends on the underlying transport provider (it is
2624		often T_INFINITE). Legal values for this field are T_UNSPEC,
2625		T_INFINITE and all non-negative numbers.
2626		The <i>l_linger</i> value is not an absolute requirement. The implementation
2627		may place upper and lower limits to this value. Requests that fall short of
2628		the lower limit are negotiated to the lower limit.
2629		Note that this option does not linger the execution of <i>t_snddis()</i> .
2630	XTI_RCVBUF	This option is used to adjust the internal buffer size allocated for the
2631		receive buffer. The buffer size may be increased for high-volume
2632		connections, or decreased to limit the possible backlog of incoming data.
2633		This request is not an absolute requirement. The implementation may
2634		place upper and lower limits on the option value. Requests that fall short
2635		of the lower limit are negotiated to the lower limit.
2636		Legal values are all positive numbers.
2637	XTI_RCVLOWAT	This option is used to set a low-water mark in the receive buffer. The
2638		option value gives the minimal number of bytes that must have
2639		accumulated in the receive buffer before they become visible to the
2640		transport user. If and when the amount of accumulated receive data
2641		exceeds the low-water mark, a T_DATA event is created, an event
2642		mechanism (for example, <i>poll()</i> or <i>select()</i>) indicates the data, and the
2643		data can be read by <i>t_rcv()</i> or <i>t_rcvudata()</i> .
2644		This request is not an absolute requirement. The implementation may
2645		place upper and lower limits on the option value. Requests that fall short
2646		of the lower limit are negotiated to the lower limit.
2647		Legal values are all positive numbers.
2648	XTI_SNDBUF	This option is used to adjust the internal buffer size allocated for the send
2649		buffer.
2650		This request is not an absolute requirement. The implementation may
2651		place upper and lower limits on the option value. Requests that fall short
2652		of the lower limit are negotiated to the lower limit.
2653		Legal values are all positive numbers.
2654	XTI_SNDLOWAT	This option is used to set a low-water mark in the send buffer. The option
2655		value gives the minimal number of bytes that must have accumulated in
2656		the send buffer before they are sent.
2657		This request is not an absolute requirement. The implementation may
2658		place upper and lower limits on the option value. Requests that fall short
2659		of the lower limit are negotiated to the lower limit.

2660 Legal values are all positive numbers.

2661 **VALID STATES**

2662 ALL - apart from T_UNINIT

2663 **ERRORS**

2664 On failure, *t_errno* is set to one of the following:

2665 [TBADF] The specified file descriptor does not refer to a transport endpoint.

2666 [TOUTSTATE] The function was issued in the wrong sequence.

2667 [TACCES] The user does not have permission to negotiate the specified options.

2668 [TBADOPT] The specified options were in an incorrect format or contained illegal
2669 information.

2670 [TBADFLAG] An invalid flag was specified.

2671 [TBUFOVFLW] The number of bytes allowed for an incoming argument (*maxlen*) is
2672 greater than 0 but not sufficient to store the value of that argument. The
2673 information to be returned in *ret* will be discarded.

2674 [TSYSERR] A system error has occurred during execution of this function.

2675 [TPROTO] This error indicates that a communication problem has been detected
2676 between XTI and the transport provider for which there is no other
2677 suitable XTI (*t_errno*).

2678 [TNOTSUPPORT] This action is not supported by the transport provider.

2679 **RETURN VALUE**

2680 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
2681 *t_errno* is set to indicate an error.

2682 **SEE ALSO**

2683 *t_accept()*, *t_alloc()*, *t_connect()*, *t_getinfo()*, *t_listen()*, *t_open()*, *t_rcvconnect()*, Chapter 6.

2684 **CHANGE HISTORY**

2685 **Issue 4**

2686 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2687 **NAME**

2688 t_rcv - receive data or expedited data sent over a connection

2689 **SYNOPSIS**

2690 #include <xti.h>

2691 int t_rcv(int *fd*, char **buf*, unsigned int *nbytes*, int **flags*);2692 **DESCRIPTION**2693
2694

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

2699 This function receives either normal or expedited data. The argument *fd* identifies the local
2700 transport endpoint through which data will arrive, *buf* points to a receive buffer where user data
2701 will be placed, and *nbytes* specifies the size of the receive buffer. The argument *flags* may be set
2702 on return from *t_rcv()* and specifies optional flags as described below.

2703 By default, *t_rcv()* operates in synchronous mode and will wait for data to arrive if none is
2704 currently available. However, if O_NONBLOCK is set (via *t_open()* or *fcntl()*), *t_rcv()* will
2705 execute in asynchronous mode and will fail if no data is available. (See [TNODATA] below.)

2706 On return from the call, if T_MORE is set in *flags*, this indicates that there is more data, and the
2707 current transport service data unit (TSDU) or expedited transport service data unit (ETSDU)
2708 must be received in multiple *t_rcv()* calls. In the asynchronous mode, the T_MORE flag may be
2709 set on return from the *t_rcv()* call even when the number of bytes received is less than the size of
2710 the receive buffer specified. Each *t_rcv()* with the T_MORE flag set indicates that another
2711 *t_rcv()* must follow to get more data for the current TSDU. The end of the TSDU is identified by
2712 the return of a *t_rcv()* call with the T_MORE flag not set. If the transport provider does not
2713 support the concept of a TSDU as indicated in the *info* argument on return from *t_open()* or
2714 *t_getinfo()*, the T_MORE flag is not meaningful and should be ignored. If *nbytes* is greater than
2715 zero on the call to *t_rcv()*, *t_rcv()* will return 0 only if the end of a TSDU is being returned to the
2716 user.

2717 On return, the data returned is expedited data if T_EXPEDITED is set in *flags*. If the number of
2718 bytes of expedited data exceeds *nbytes*, *t_rcv()* will set T_EXPEDITED and T_MORE on return
2719 from the initial call. Subsequent calls to retrieve the remaining ETSDU will have T_EXPEDITED
2720 set on return. The end of the ETSDU is identified by the return of a *t_rcv()* call with the
2721 T_MORE flag not set.

2722 In synchronous mode, the only way for the user to be notified of the arrival of normal or
2723 expedited data is to issue this function or check for the T_DATA or T_EXDATA events using the
2724 *t_look()* function. Additionally, the process can arrange to be notified via the EM interface.

2725 **VALID STATES**

2726 T_DATAXFER, T_OUTREL

2727 **ERRORS**2728 On failure, *t_errno* is set to one of the following:

2729 [TBADF] The specified file descriptor does not refer to a transport endpoint.

2730 [TNODATA] O_NONBLOCK was set, but no data is currently available from the
2731 transport provider.

2732	[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
2733		
2734	[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
2735	[TOUTSTATE]	The function was issued in the wrong sequence on the transport endpoint referenced by <i>fd</i> .
2736		
2737	[TSYSERR]	A system error has occurred during execution of this function.
2738	[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (<i>t_errno</i>).
2739		
2740		

2741 RETURN VALUE

2742 On successful completion, *t_rcv()* returns the number of bytes received. Otherwise, it returns -1
2743 on failure and *t_errno* is set to indicate the error.

2744 SEE ALSO

2745 *fcntl()*, *t_getinfo()*, *t_look()*, *t_open()*, *t_snd()*.

2746 CHANGE HISTORY**2747 Issue 4**

2748 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2749 **NAME**

2750 `t_rcvconnect` - receive the confirmation from a connect request

2751 **SYNOPSIS**

2752 `#include <xti.h>`

2753 `int t_rcvconnect(int fd, struct t_call *call);`

2754 **DESCRIPTION**

2755
2756

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

2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767

2768 This function enables a calling transport user to determine the status of a previously sent
2769 connect request and is used in conjunction with `t_connect()` to establish a connection in
2770 asynchronous mode. The connection will be established on successful completion of this
2771 function.

2772 The argument *fd* identifies the local transport endpoint where communication will be
2773 established, and *call* contains information associated with the newly established connection. The
2774 argument *call* points to a `t_call` structure which contains the following members:

```
2775     struct netbuf addr;
2776     struct netbuf opt;
2777     struct netbuf udata;
2778     int sequence;
```

2779 In *call*, *addr* returns the protocol address associated with the responding transport endpoint, *opt*
2780 presents any options associated with the connection, *udata* points to optional user data that may
2781 be returned by the destination transport user during connection establishment, and *sequence* has
2782 no meaning for this function.

2783 The *maxlen* field of each argument must be set before issuing this function to indicate the
2784 maximum size of the buffer for each. However, *call* may be a null pointer, in which case no
2785 information is given to the user on return from `t_rcvconnect()`. By default, `t_rcvconnect()`
2786 executes in synchronous mode and waits for the connection to be established before returning.
2787 On return, the *addr*, *opt* and *udata* fields reflect values associated with the connection.

2788 If `O_NONBLOCK` is set (via `t_open()` or `fcntl()`), `t_rcvconnect()` executes in asynchronous mode,
2789 and reduces to a poll for existing connect confirmations. If none are available, `t_rcvconnect()`
2790 fails and returns immediately without waiting for the connection to be established. (See
2791 `[TNODATA]` below.) In this case, `t_rcvconnect()` must be called again to complete the connection
2792 establishment phase and retrieve the information returned in *call*.

2793 **VALID STATES**

2794 `T_OUTCON`

2795 **ERRORS**

2796 On failure, *t_errno* is set to one of the following:

2797	[TBADF]	The specified file descriptor does not refer to a transport endpoint.
2798	[TBUFOVFLW]	The number of bytes allocated for an incoming argument (<i>maxlen</i>) is greater than 0 but not sufficient to store the value of that argument, and the connect information to be returned in <i>call</i> will be discarded. The provider's state, as seen by the user, will be changed to T_DATAXFER.
2799		
2800		
2801		
2802	[TNODATA]	O_NONBLOCK was set, but a connect confirmation has not yet arrived.
2803	[TLOOK]	An asynchronous event has occurred on this transport connection and requires immediate attention.
2804		
2805	[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
2806	[TOUTSTATE]	The function was issued in the wrong sequence on the transport endpoint referenced by <i>fd</i> .
2807		
2808	[TSYSERR]	A system error has occurred during execution of this function.
2809	[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (<i>t_errno</i>).
2810		
2811		

2812 **RETURN VALUE**

2813 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
2814 *t_errno* is set to indicate an error.

2815 **SEE ALSO**

2816 *t_accept()*, *t_alloc()*, *t_bind()*, *t_connect()*, *t_listen()*, *t_open()*, *t_optmgmt()*.

2817 **CHANGE HISTORY**2818 **Issue 4**

2819 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2820 **NAME**

2821 t_rcvdis - retrieve information from disconnect

2822 **SYNOPSIS**

2823 #include <xti.h>

2824 int t_rcvdis(int *fd*, struct t_discon **discon*);2825 **DESCRIPTION**

2826

2827

Parameters	Before call	After call
<i>fd</i>	x	/
<i>discon->udata.maxlen</i>	x	/
<i>discon->udata.len</i>	/	x
<i>discon->udata.buf</i>	?	(?)
<i>discon->reason</i>	/	x
<i>discon->sequence</i>	/	?

2828

2829

2830

2831

2832

2833

2834 This function is used to identify the cause of a disconnect and to retrieve any user data sent with
 2835 the disconnect. The argument *fd* identifies the local transport endpoint where the connection
 2836 existed, and *discon* points to a **t_discon** structure containing the following members:

2837 struct netbuf *udata*;2838 int *reason*;2839 int *sequence*;

2840 The field *reason* specifies the reason for the disconnect through a protocol-dependent reason
 2841 code, *udata* identifies any user data that was sent with the disconnect, and *sequence* may identify
 2842 an outstanding connect indication with which the disconnect is associated. The field *sequence*
 2843 is only meaningful when *t_rcvdis()* is issued by a passive transport user who has executed one or
 2844 more *t_listen()* functions and is processing the resulting connect indications. If a disconnect
 2845 indication occurs, *sequence* can be used to identify which of the outstanding connect indications
 2846 is associated with the disconnect.

2847 If a user does not care if there is incoming data and does not need to know the value of *reason* or
 2848 *sequence*, *discon* may be a null pointer and any user data associated with the disconnect will be
 2849 discarded. However, if a user has retrieved more than one outstanding connect indication (via
 2850 *t_listen()*) and *discon* is a null pointer, the user will be unable to identify with which connect
 2851 indication the disconnect is associated.

2852 **VALID STATES**2853 T_DATAXFER,T_OUTCON,T_OUTREL,T_INREL,T_INCON(*ocnt* > 0)2854 **ERRORS**2855 On failure, *t_errno* is set to one of the following:

2856 [TBADF] The specified file descriptor does not refer to a transport endpoint.

2857 [TNODIS] No disconnect indication currently exists on the specified transport
2858 endpoint.2859 [TBUFOVFLW] The number of bytes allocated for incoming data (*maxlen*) is greater than 0
2860 but not sufficient to store the data. If *fd* is a passive endpoint with *ocnt* >
2861 1, it remains in state T_INCON; otherwise, the endpoint state is set to
2862 T_IDLE.

2863 [TNOTSUPPORT] This function is not supported by the underlying transport provider.

- 2864 [TSYSERR] A system error has occurred during execution of this function.
- 2865 [TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint
2866 referenced by *fd*.
- 2867 [TPROTO] This error indicates that a communication problem has been detected
2868 between XTI and the transport provider for which there is no other
2869 suitable XTI (*t_errno*).
- 2870 **RETURN VALUE**
- 2871 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
2872 *t_errno* is set to indicate an error.
- 2873 **SEE ALSO**
- 2874 *t_alloc()*, *t_connect()*, *t_listen()*, *t_open()*, *t_snddis()*.
- 2875 **CHANGE HISTORY**
- 2876 **Issue 4**
- 2877 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2878 **NAME**

2879 t_rcvrel - acknowledge receipt of an orderly release indication

2880 **SYNOPSIS**

2881 #include <xti.h>

2882 int t_rcvrel(int *fd*);2883 **DESCRIPTION**

2884

2885

2886

Parameters	Before call	After call
<i>fd</i>	x	/

2887 This function is used to acknowledge receipt of an orderly release indication. The argument *fd*
 2888 identifies the local transport endpoint where the connection exists. After receipt of this
 2889 indication, the user may not attempt to receive more data because such an attempt will block
 2890 forever. However, the user may continue to send data over the connection if *t_sndrel()* has not
 2891 been called by the user. This function is an optional service of the transport provider, and is
 2892 only supported if the transport provider returned service type T_COTS_ORD on *t_open()* or
 2893 *t_getinfo()*.

2894 **VALID STATES**

2895 T_DATAXFER,T_OUTREL

2896 **ERRORS**2897 On failure, *t_errno* is set to one of the following:

2898 [TBADF]	The specified file descriptor does not refer to a transport endpoint.
2899 [TNOREL]	No orderly release indication currently exists on the specified transport endpoint.
2900	
2901 [TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
2902	
2903 [TNOTSUPPORT]	This function is not supported by the underlying transport provider.
2904 [TSYSERR]	A system error has occurred during execution of this function.
2905 [TOUTSTATE]	The function was issued in the wrong sequence on the transport endpoint referenced by <i>fd</i> .
2906	
2907 [TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (<i>t_errno</i>).
2908	
2909	

2910 **RETURN VALUE**

2911 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
 2912 *t_errno* is set to indicate an error.

2913 **SEE ALSO**2914 *t_getinfo()*, *t_open()*, *t_sndrel()*.2915 **CHANGE HISTORY**2916 **Issue 4**2917 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2918 **NAME**

2919 t_rcvudata - receive a data unit

2920 **SYNOPSIS**

2921 #include <xti.h>

2922 int t_rcvudata(int *fd*, struct t_unitdata **unitdata*, int **flags*);2923 **DESCRIPTION**2924
2925

2926

2927

2928

2929

2930

2931

2932

2933

2934

2935

2936

Parameters	Before call	After call
<i>fd</i>	x	/
<i>unitdata</i> -> <i>addr.maxlen</i>	x	/
<i>unitdata</i> -> <i>addr.len</i>	/	x
<i>unitdata</i> -> <i>addr.buf</i>	?	(?)
<i>unitdata</i> -> <i>opt.maxlen</i>	x	/
<i>unitdata</i> -> <i>opt.len</i>	/	x
<i>unitdata</i> -> <i>opt.buf</i>	?	(?)
<i>unitdata</i> -> <i>udata.maxlen</i>	x	/
<i>unitdata</i> -> <i>udata.len</i>	/	x
<i>unitdata</i> -> <i>udata.buf</i>	?	(?)
<i>flags</i>	/	x

2937 This function is used in connectionless mode to receive a data unit from another transport user.
 2938 The argument *fd* identifies the local transport endpoint through which data will be received,
 2939 *unitdata* holds information associated with the received data unit, and *flags* is set on return to
 2940 indicate that the complete data unit was not received. The argument *unitdata* points to a
 2941 **t_unitdata** structure containing the following members:

2942 struct netbuf *addr*;2943 struct netbuf *opt*;2944 struct netbuf *udata*;

2945 The *maxlen* field of *addr*, *opt* and *udata* must be set before calling this function to indicate the
 2946 maximum size of the buffer for each.

2947 On return from this call, *addr* specifies the protocol address of the sending user, *opt* identifies
 2948 options that were associated with this data unit, and *udata* specifies the user data that was
 2949 received.

2950 By default, *t_rcvudata*() operates in synchronous mode and will wait for a data unit to arrive if
 2951 none is currently available. However, if O_NONBLOCK is set (via *t_open*() or *fcntl*()),
 2952 *t_rcvudata*() will execute in asynchronous mode and will fail if no data units are available.

2953 If the buffer defined in the *udata* field of *unitdata* is not large enough to hold the current data
 2954 unit, the buffer will be filled and T_MORE will be set in *flags* on return to indicate that another
 2955 *t_rcvudata*() should be called to retrieve the rest of the data unit. Subsequent calls to
 2956 *t_rcvudata*() will return zero for the length of the address and options until the full data unit has
 2957 been received.

2958 **VALID STATES**

2959 T_IDLE

2960 **ERRORS**

2961 On failure, *t_errno* is set to one of the following:

2962	[TBADF]	The specified file descriptor does not refer to a transport endpoint.
2963	[TNODATA]	O_NONBLOCK was set, but no data units are currently available from the transport provider.
2964		
2965	[TBUFOVFLW]	The number of bytes allocated for the incoming protocol address or options (<i>maxlen</i>) is greater than 0 but not sufficient to store the information. The unit data information to be returned in <i>unitdata</i> will be discarded.
2966		
2967		
2968		
2969	[TLOOK]	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
2970		
2971	[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
2972	[TOUTSTATE]	The function was issued in the wrong sequence on the transport endpoint referenced by <i>fd</i> .
2973		
2974	[TSYSERR]	A system error has occurred during execution of this function.
2975	[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (<i>t_errno</i>).
2976		
2977		

2978 **RETURN VALUE**

2979 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
2980 *t_errno* is set to indicate an error.

2981 **SEE ALSO**

2982 *fcntl()*, *t_alloc()*, *t_open()*, *t_rcvuderr()*, *t_sndudata()*.

2983 **CHANGE HISTORY**2984 **Issue 4**

2985 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

2986 **NAME**

2987 t_rcvuderr - receive a unit data error indication

2988 **SYNOPSIS**

2989 #include <xti.h>

2990 int t_rcvuderr(int *fd*, struct t_uderr **uderr*);2991 **DESCRIPTION**2992
2993

Parameters	Before call	After call
<i>fd</i>	x	/
<i>uderr</i> -> <i>addr.maxlen</i>	x	/
<i>uderr</i> -> <i>addr.len</i>	/	x
<i>uderr</i> -> <i>addr.buf</i>	?	(?)
<i>uderr</i> -> <i>opt.maxlen</i>	x	/
<i>uderr</i> -> <i>opt.len</i>	/	x
<i>uderr</i> -> <i>opt.buf</i>	?	(?)
<i>uderr</i> -> <i>error</i>	/	x

3001

3002 This function is used in connectionless mode to receive information concerning an error on a
 3003 previously sent data unit, and should only be issued following a unit data error indication. It
 3004 informs the transport user that a data unit with a specific destination address and protocol
 3005 options produced an error. The argument *fd* identifies the local transport endpoint through
 3006 which the error report will be received, and *uderr* points to a **t_uderr** structure containing the
 3007 following members:

```
3008     struct netbuf addr;
3009     struct netbuf opt;
3010     long error;
```

3011 The *maxlen* field of *addr* and *opt* must be set before calling this function to indicate the maximum
 3012 size of the buffer for each.

3013 On return from this call, the *addr* structure specifies the destination protocol address of the
 3014 erroneous data unit, the *opt* structure identifies options that were associated with the data unit,
 3015 and *error* specifies a protocol-dependent error code.

3016 If the user does not care to identify the data unit that produced an error, *uderr* may be set to a
 3017 null pointer, and *t_rcvuderr()* will simply clear the error indication without reporting any
 3018 information to the user.

3019 **VALID STATES**

3020 T_IDLE

3021 **ERRORS**3022 On failure, *t_errno* is set to one of the following:

3023 [TBADF] The specified file descriptor does not refer to a transport endpoint.

3024 [TNOUDERR] No unit data error indication currently exists on the specified transport
 3025 endpoint.

3026 [TBUFOVFLW] The number of bytes allocated for the incoming protocol address or
 3027 options (*maxlen*) is greater than 0 but not sufficient to store the
 3028 information. The unit data error information to be returned in *uderr* will
 3029 be discarded.

- 3030 [TNOTSUPPORT] This function is not supported by the underlying transport provider.
- 3031 [TSYSERR] A system error has occurred during execution of this function.
- 3032 [TPROTO] This error indicates that a communication problem has been detected
3033 between XTI and the transport provider for which there is no other
3034 suitable XTI (*t_errno*).
- 3035 **RETURN VALUE**
- 3036 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
3037 *t_errno* is set to indicate an error.
- 3038 **SEE ALSO**
- 3039 *t_rcvudata()*, *t_sndudata()*.
- 3040 **CHANGE HISTORY**
- 3041 **Issue 4**
- 3042 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

3043 **NAME**

3044 t_snd - send data or expedited data over a connection

3045 **SYNOPSIS**

3046 #include <xti.h>

3047 int t_snd(int fd, char *buf, unsigned int nbytes, int flags);

3048 **DESCRIPTION**3049
3050

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

3051
3052
3053
3054

3055 This function is used to send either normal or expedited data. The argument *fd* identifies the
3056 local transport endpoint over which data should be sent, *buf* points to the user data, *nbytes*
3057 specifies the number of bytes of user data to be sent, and *flags* specifies any optional flags
3058 described below:

3059 **T_EXPEDITED** If set in *flags*, the data will be sent as expedited data and will be subject to
3060 the interpretations of the transport provider.

3061 **T_MORE** If set in *flags*, this indicates to the transport provider that the transport
3062 service data unit (TSDU) (or expedited transport service data unit -
3063 ETSDU) is being sent through multiple *t_snd()* calls. Each *t_snd()* with
3064 the T_MORE flag set indicates that another *t_snd()* will follow with more
3065 data for the current TSDU (or ETSDU).

3066 The end of the TSDU (or ETSDU) is identified by a *t_snd()* call with the
3067 T_MORE flag not set. Use of T_MORE enables a user to break up large
3068 logical data units without losing the boundaries of those units at the other
3069 end of the connection. The flag implies nothing about how the data is
3070 packaged for transfer below the transport interface. If the transport
3071 provider does not support the concept of a TSDU as indicated in the *info*
3072 argument on return from *t_open()* or *t_getinfo()*, the T_MORE flag is not
3073 meaningful and will be ignored if set.

3074 The sending of a zero-length fragment of a TSDU or ETSDU is only
3075 permitted where this is used to indicate the end of a TSDU or ETSDU;
3076 that is, when the T_MORE flag is not set. Some transport providers also
3077 forbid zero-length TSDUs and ETSDUs. See Appendix A on page 189 for
3078 a fuller explanation.

3079 By default, *t_snd()* operates in synchronous mode and may wait if flow control restrictions
3080 prevent the data from being accepted by the local transport provider at the time the call is made.
3081 However, if O_NONBLOCK is set (via *t_open()* or *fcntl()*), *t_snd()* will execute in asynchronous
3082 mode, and will fail immediately if there are flow control restrictions. The process can arrange to
3083 be informed when the flow control restrictions are cleared via either *t_look()* or the EM interface.

3084 On successful completion, *t_snd()* returns the number of bytes accepted by the transport
3085 provider. Normally this will equal the number of bytes specified in *nbytes*. However, if
3086 O_NONBLOCK is set, it is possible that only part of the data will actually be accepted by the
3087 transport provider. In this case, *t_snd()* will return a value that is less than the value of *nbytes*. If
3088 *nbytes* is zero and sending of zero octets is not supported by the underlying transport service,
3089 *t_snd()* will return -1 with *t_errno* set to [TBADDDATA].

3090 The size of each TSDU or ETSDU must not exceed the limits of the transport provider as
 3091 specified by the current values in the TSDU or ETSDU fields in the *info* argument returned by
 3092 *t_getinfo()*.

3093 The error [TLOOK] may be returned to inform the process that an event (for example, a
 3094 disconnect) has occurred.

3095 VALID STATES

3096 T_DATAXFER, T_INREL

3097 ERRORS

3098 On failure, *t_errno* is set to one of the following:

3099 [TBADF] The specified file descriptor does not refer to a transport endpoint.

3100 [TBADDATA] Illegal amount of data:

3101 — A single send was attempted specifying a TSDU (ETSDU) or fragment
 3102 TSDU (ETSDU) greater than that specified by the current values of the
 3103 TSDU or ETSDU fields in the *info* argument.

3104 — A send of a zero byte TSDU (ETSDU) or zero byte fragment of a TSDU
 3105 (ETSDU) is not supported by the provider (see Appendix A on page
 3106 189).

3107 — Multiple sends were attempted resulting in a TSDU (ETSDU) larger
 3108 than that specified by the current value of the TSDU or ETSDU fields
 3109 in the *info* argument — the ability of an XTI implementation to detect
 3110 such an error case is implementation-dependent (see **CAVEATS**,
 3111 below).

3112 [TBADFLAG] An invalid flag was specified.

3113 [TFLOW] O_NONBLOCK was set, but the flow control mechanism prevented the
 3114 transport provider from accepting any data at this time.

3115 [TNOTSUPPORT] This function is not supported by the underlying transport provider.

3116 [TLOOK] An asynchronous event has occurred on this transport endpoint.

3117 [TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint
 3118 referenced by *fd*.

3119 [TSYSERR] A system error has occurred during execution of this function.

3120 [TPROTO] This error indicates that a communication problem has been detected
 3121 between XTI and the transport provider for which there is no other
 3122 suitable XTI (*t_errno*).

3123 RETURN VALUE

3124 On successful completion, *t_snd()* returns the number of bytes accepted by the transport
 3125 provider. Otherwise, -1 is returned on failure and *t_errno* is set to indicate the error.

3126 Note that in asynchronous mode, if the number of bytes accepted by the transport provider is
 3127 less than the number of bytes requested, this may indicate that the transport provider is blocked
 3128 due to flow control.

3129 SEE ALSO

3130 *t_getinfo()*, *t_open()*, *t_rcv()*.

3131 **CAVEATS**

3132 It is important to remember that the transport provider treats all users of a transport endpoint as
3133 a single user. Therefore if several processes issue concurrent *t_snd()* calls then the different data
3134 may be intermixed.

3135 Multiple sends which exceed the maximum TSDU or ETSDU size may not be discovered by XTI.
3136 In this case an implementation-dependent error will result (generated by the transport provider)
3137 perhaps on a subsequent XTI call. This error may take the form of a connection abort, a
3138 [TSYSERR], a [TBADDDATA] or a [TPROTO] error.

3139 If multiple sends which exceed the maximum TSDU or ETSDU size are detected by XTI, *t_snd()*
3140 fails with [TBADDDATA].

3141 **CHANGE HISTORY**3142 **Issue 4**

3143 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

3144 NAME

3145 t_snddis - send user-initiated disconnect request

3146 SYNOPSIS

3147 #include <xti.h>

3148 int t_snddis(int *fd*, struct t_call **call*);

3149 DESCRIPTION

3150

3151

3152

3153

3154

3155

3156

3157

3158

3159

3160

3161

3162

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

3163

3164

3165

3166

This function is used to initiate an abortive release on an already established connection, or to reject a connect request. The argument *fd* identifies the local transport endpoint of the connection, and *call* specifies information associated with the abortive release. The argument *call* points to a **t_call** structure which contains the following members:

3167

3168

3169

3170

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

3171

3172

3173

3174

3175

3176

3177

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.

3178

3179

3180

3181

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.

3182 VALID STATES

3183 T_DATAXFER,T_OUTCON,T_OUTREL,T_INREL,T_INCON(ocnt > 0)

3184 ERRORS

3185 On failure, *t_errno* is set to one of the following:

3186

3187

3188

[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*.

3189	[TBADDATA]	The amount of user data specified was not within the bounds allowed by the transport provider.
3190		
3191	[TBADSEQ]	An invalid sequence number was specified, or a null <i>call</i> pointer was specified, when rejecting a connect request.
3192		
3193	[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
3194	[TSYSERR]	A system error has occurred during execution of this function.
3195	[TLOOK]	An asynchronous event, which requires attention, has occurred.
3196	[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (<i>t_errno</i>).
3197		
3198		
3199	RETURN VALUE	
3200		Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and <i>t_errno</i> is set to indicate an error.
3201		
3202	SEE ALSO	
3203		<i>t_connect()</i> , <i>t_getinfo()</i> , <i>t_listen()</i> , <i>t_open()</i> .
3204	CAVEATS	
3205		<i>t_snddis()</i> is an abortive disconnect. Therefore a <i>t_snddis()</i> issued on a connection endpoint may cause data previously sent via <i>t_snd()</i> , or data not yet received, to be lost (even if an error is returned).
3206		
3207		
3208	CHANGE HISTORY	
3209	Issue 4	
3210		The SYNOPSIS section is placed in the form of a standard C function prototype.

3211 **NAME**

3212 t_sndrel - initiate an orderly release

3213 **SYNOPSIS**

3214 #include <xti.h>

3215 int t_sndrel(int *fd*);3216 **DESCRIPTION**

3217

3218

3219

Parameters	Before call	After call
<i>fd</i>	x	/

3220 This function is used to initiate an orderly release of a transport connection and indicates to the
 3221 transport provider that the transport user has no more data to send. The argument *fd* identifies
 3222 the local transport endpoint where the connection exists. After calling *t_sndrel()*, the user may
 3223 not send any more data over the connection. However, a user may continue to receive data if an
 3224 orderly release indication has not been received. This function is an optional service of the
 3225 transport provider and is only supported if the transport provider returned service type
 3226 T_COTS_ORD on *t_open()* or *t_getinfo()*.

3227 **VALID STATES**

3228 T_DATAXFER,T_INREL

3229 **ERRORS**3230 On failure, *t_errno* is set to one of the following:

3231 [TBADF]	The specified file descriptor does not refer to a transport endpoint.
3232 [TFLOW]	O_NONBLOCK was set, but the flow control mechanism prevented the
3233	transport provider from accepting the function at this time.
3234 [TLOOK]	An asynchronous event has occurred on this transport endpoint and
3235	requires immediate attention.
3236 [TNOTSUPPORT]	This function is not supported by the underlying transport provider.
3237 [TOUTSTATE]	The function was issued in the wrong sequence on the transport endpoint
3238	referenced by <i>fd</i> .
3239 [TSYSERR]	A system error has occurred during execution of this function.
3240 [TPROTO]	This error indicates that a communication problem has been detected
3241	between XTI and the transport provider for which there is no other
3242	suitable XTI (<i>t_errno</i>).

3243 **RETURN VALUE**

3244 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
 3245 *t_errno* is set to indicate an error.

3246 **SEE ALSO**3247 *t_getinfo()*, *t_open()*, *t_rcvrel()*.3248 **CHANGE HISTORY**3249 **Issue 4**3250 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

3251 NAME

3252 t_sndudata - send a data unit

3253 SYNOPSIS

3254 #include <xti.h>

3255 int t_sndudata(int *fd*, struct t_unitdata **unitdata*);

3256 DESCRIPTION

3257

3258

3259

3260

3261

3262

3263

3264

3265

3266

3267

3268

Parameters	Before call	After call
<i>fd</i>	x	/
<i>unitdata</i> -> <i>addr.maxlen</i>	/	/
<i>unitdata</i> -> <i>addr.len</i>	x	/
<i>unitdata</i> -> <i>addr.buf</i>	x(x)	/
<i>unitdata</i> -> <i>opt.maxlen</i>	/	/
<i>unitdata</i> -> <i>opt.len</i>	x	/
<i>unitdata</i> -> <i>opt.buf</i>	?(?)	/
<i>unitdata</i> -> <i>udata.maxlen</i>	/	/
<i>unitdata</i> -> <i>udata.len</i>	x	/
<i>unitdata</i> -> <i>udata.buf</i>	x(x)	/

3269 This function is used in connectionless mode to send a data unit to another transport user. The
 3270 argument *fd* identifies the local transport endpoint through which data will be sent, and *unitdata*
 3271 points to a **t_unitdata** structure containing the following members:

3272 struct netbuf *addr*;3273 struct netbuf *opt*;3274 struct netbuf *udata*;

3275 In *unitdata*, *addr* specifies the protocol address of the destination user, *opt* identifies options that
 3276 the user wants associated with this request, and *udata* specifies the user data to be sent. The user
 3277 may choose not to specify what protocol options are associated with the transfer by setting the
 3278 *len* field of *opt* to zero. In this case, the provider may use default options.

3279 If the *len* field of *udata* is zero, and sending of zero octets is not supported by the underlying
 3280 transport service, the *t_sndudata()* will return -1 with *t_errno* set to [TBADDDATA].

3281 By default, *t_sndudata()* operates in synchronous mode and may wait if flow control restrictions
 3282 prevent the data from being accepted by the local transport provider at the time the call is made.
 3283 However, if O_NONBLOCK is set (via *t_open()* or *fcntl()*), *t_sndudata()* will execute in
 3284 asynchronous mode and will fail under such conditions. The process can arrange to be notified
 3285 of the clearance of a flow control restriction via either *t_look()* or the EM interface.

3286 If the amount of data specified in *udata* exceeds the TSDU size as returned in the *tsdu* field of the
 3287 *info* argument of *t_open()* or *t_getinfo()*, a [TBADDDATA] error will be generated. If *t_sndudata()*
 3288 is called before the destination user has activated its transport endpoint (see *t_bind()*), the data
 3289 unit may be discarded.

3290 If it is not possible for the transport provider to immediately detect the conditions that cause the
 3291 errors [TBADDDADDR] and [TBADDOPT]. These errors will alternatively be returned by
 3292 *t_rcvuderr*. Therefore, an application must be prepared to receive these errors in both of these
 3293 ways.

3294 VALID STATES

3295 T_IDLE

3296 **ERRORS**3297 On failure, *t_errno* is set to one of the following:3298 [TBADDDATA] Illegal amount of data. A single send was attempted specifying a TSDU
3299 greater than that specified in the *info* argument, or a send of a zero byte
3300 TSDU is not supported by the provider.

3301 [TBADF] The specified file descriptor does not refer to a transport endpoint.

3302 [TFLOW] O_NONBLOCK was set, but the flow control mechanism prevented the
3303 transport provider from accepting any data at this time.

3304 [TLOOK] An asynchronous event has occurred on this transport endpoint.

3305 [TNOTSUPPORT] This function is not supported by the underlying transport provider.

3306 [TOUTSTATE] The function was issued in the wrong sequence on the transport endpoint
3307 referenced by *fd*.

3308 [TSYSERR] A system error has occurred during execution of this function.

3309 [TBADADDR] The specified protocol address was in an incorrect format or contained
3310 illegal information.3311 [TBADOPT] The specified options were in an incorrect format or contained illegal
3312 information.3313 [TPROTO] This error indicates that a communication problem has been detected
3314 between XTI and the transport provider for which there is no other
3315 suitable XTI (*t_errno*).3316 **RETURN VALUE**3317 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
3318 *t_errno* is set to indicate an error.3319 **SEE ALSO**3320 *fcntl()*, *t_alloc()*, *t_open()*, *t_rcvudata()*, *t_rcvuderr()*.3321 **CHANGE HISTORY**3322 **Issue 4**3323 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

3324 **NAME**

3325 t_strerror - produce an error message string

3326 **SYNOPSIS**

3327 #include <xti.h>

3328 char *t_strerror(int *errnum*);3329 **DESCRIPTION**

3330

3331

3332

Parameters	Before call	After call
<i>errnum</i>	x	/

3333 The *t_strerror()* function maps the error number in *errnum* that corresponds to an XTI error to a language-dependent error message string and returns a pointer to the string. The string pointed to will not be modified by the program, but may be overwritten by a subsequent call to the *t_strerror* function. The string is not terminated by a newline character. The language for error message strings written by *t_strerror()* is implementation-defined. If it is English, the error message string describing the value in *t_errno* is identical to the comments following the *t_errno* codes defined in <xti.h>. If an error code is unknown, and the language is English, *t_strerror()* returns the string:

3341 "<error>: error unknown"

3342 where <error> is the error number supplied as input. In other languages, an equivalent text is provided.

3343

3344 **VALID STATES**

3345 ALL - apart from T_UNINIT

3346 **RETURN VALUE**3347 The function *t_strerror()* returns a pointer to the generated message string.3348 **SEE ALSO**3349 *t_error()*3350 **CHANGE HISTORY**3351 **Issue 4**3352 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

3353 **NAME**

3354 t_sync - synchronise transport library

3355 **SYNOPSIS**

```
3356 #include <xti.h>
3357 int t_sync(int fd);
```

3358 **DESCRIPTION**

3359

Parameters	Before call	After call
<i>fd</i>	x	/

3361

3362 For the transport endpoint specified by *fd*, *t_sync()* synchronises the data structures managed by
 3363 the transport library with information from the underlying transport provider. In doing so, it
 3364 can convert an uninitialised file descriptor (obtained via *open()*, *dup()* or as a result of a *fork()*
 3365 and *exec()*) to an initialised transport endpoint, assuming that the file descriptor referenced a
 3366 transport endpoint, by updating and allocating the necessary library data structures. This
 3367 function also allows two cooperating processes to synchronise their interaction with a transport
 3368 provider.

3369 For example, if a process forks a new process and issues an *exec()*, the new process must issue a
 3370 *t_sync()* to build the private library data structure associated with a transport endpoint and to
 3371 synchronise the data structure with the relevant provider information.

3372 It is important to remember that the transport provider treats all users of a transport endpoint as
 3373 a single user. If multiple processes are using the same endpoint, they should coordinate their
 3374 activities so as not to violate the state of the transport endpoint. The function *t_sync()* returns
 3375 the current state of the transport endpoint to the user, thereby enabling the user to verify the
 3376 state before taking further action. This coordination is only valid among cooperating processes;
 3377 it is possible that a process or an incoming event could change the endpoint's state *after* a
 3378 *t_sync()* is issued.

3379 If the transport endpoint is undergoing a state transition when *t_sync()* is called, the function
 3380 will fail.

3381 **VALID STATES**

3382 ALL - apart from T_UNINIT

3383 **ERRORS**3384 On failure, *t_errno* is set to one of the following:

3385 [TBADF] The specified file descriptor does not refer to a transport endpoint. This
 3386 error may be returned when the *fd* has been previously closed or an
 3387 erroneous number may have been passed to the call.

3388 [TSTATECHNG] The transport endpoint is undergoing a state change.

3389 [TSYSERR] A system error has occurred during execution of this function.

3390 [TPROTO] This error indicates that a communication problem has been detected
 3391 between XTI and the transport provider for which there is no other
 3392 suitable XTI (*t_errno*).

3393 **RETURN VALUE**

3394 On successful completion, the state of the transport endpoint is returned. Otherwise, a value of
 3395 -1 is returned and *t_errno* is set to indicate an error. The state returned is one of the following:

3396 T_UNBND Unbound.

3397	T_IDLE	Idle.
3398	T_OUTCON	Outgoing connection pending.
3399	T_INCON	Incoming connection pending.
3400	T_DATAXFER	Data transfer.
3401	T_OUTREL	Outgoing orderly release (waiting for an orderly release indication).
3402	T_INREL	Incoming orderly release (waiting for an orderly release request).
3403	SEE ALSO	
3404		<i>dup()</i> , <i>exec()</i> , <i>fork()</i> , <i>open()</i> .
3405	CHANGE HISTORY	
3406	Issue 4	
3407		The SYNOPSIS section is placed in the form of a standard C function prototype.

3408 **NAME**

3409 t_unbind - disable a transport endpoint

3410 **SYNOPSIS**

3411 #include <xti.h>

3412 int t_unbind(int *fd*);3413 **DESCRIPTION**

3414

3415

3416

Parameters	Before call	After call
<i>fd</i>	x	/

3417 The *t_unbind()* function disables the transport endpoint specified by *fd* which was previously
 3418 bound by *t_bind()*. On completion of this call, no further data or events destined for this
 3419 transport endpoint will be accepted by the transport provider. An endpoint which is disabled by
 3420 using *t_unbind()* can be enabled by a subsequent call to *t_bind()*.

3421 **VALID STATES**

3422 T_IDLE

3423 **ERRORS**3424 On failure, *t_errno* is set to one of the following:

3425 [TBADF] The specified file descriptor does not refer to a transport endpoint.

3426 [TOUTSTATE] The function was issued in the wrong sequence.

3427 [TLOOK] An asynchronous event has occurred on this transport endpoint.

3428 [TSYSERR] A system error has occurred during execution of this function.

3429 [TPROTO] This error indicates that a communication problem has been detected
 3430 between XTI and the transport provider for which there is no other
 3431 suitable XTI (*t_errno*).

3432 **RETURN VALUE**

3433 Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and
 3434 *t_errno* is set to indicate an error.

3435 **SEE ALSO**3436 *t_bind()*.3437 **CHANGE HISTORY**3438 **Issue 4**3439 The **SYNOPSIS** section is placed in the form of a standard C function prototype.

Sockets Interfaces

3440

3441 This chapter gives an overview of the Sockets interfaces and includes functions, macros and
3442 external variables to support portability at the C-language source level.

3443 The associated headers are documented in Chapter 9.

3444 8.1 Sockets Overview

3445 All network protocols are associated with a specific protocol family. A protocol family provides
3446 basic services to the protocol implementation to allow it to function within a specific network
3447 environment. These services can include packet fragmentation and reassembly, routing,
3448 addressing, and basic transport. A protocol family can support multiple methods of addressing,
3449 though the current protocol implementations do not. A protocol family normally comprises a
3450 number of protocols, one per socket type. It is not required that a protocol family support all
3451 socket types. A protocol family can contain multiple protocols supporting the same socket
3452 abstraction.

3453 A protocol supports one of the socket abstractions detailed in the manual page for the *socket()*
3454 function. A specific protocol can be accessed either by creating a socket of the appropriate type
3455 and protocol family, or by requesting the protocol explicitly when creating a socket. Protocols
3456 normally accept only one type of address format, usually determined by the addressing
3457 structure inherent in the design of the protocol family and network architecture. Certain
3458 semantics of the basic socket abstractions are protocol specific. All protocols are expected to
3459 support the basic model for their particular socket type, but can, in addition, provide
3460 nonstandard facilities or extensions to a mechanism. For example, a protocol supporting the
3461 SOCK_STREAM abstraction can allow more than one byte of out-of-band data to be transmitted
3462 per out-of-band message.

3463 This specification covers local UNIX connections and Internet protocols.

3464 Addressing

3465 Associated with each address family is an address format. All network addresses adhere to a
3466 general structure, called a **sockaddr**. The length of the structure varies according to the address
3467 family.

3468 Routing

3469 Sockets provides packet routing facilities. A routing information database is maintained, which
3470 is used in selecting the appropriate network interface when transmitting packets.

3471 Interfaces

3472 Each network interface in a system corresponds to a path through which messages can be sent
3473 and received. A network interface usually has a hardware device associated with it, though
3474 certain interfaces such as the loopback interface do not.

3475 **NAME**

3476 accept — accept a new connection on a socket

3477 **SYNOPSIS**3478 **UX** #include <sys/socket.h>3479 int accept (int *socket*, struct sockaddr **address*, size_t **address_len*);3480 **DESCRIPTION**3481 The *accept()* function extracts the first connection on the queue of pending connections, creates a
3482 new socket with the same socket type protocol and address family as the specified socket, and
3483 allocates a new file descriptor for that socket.

3484 The function takes the following arguments:

3485 *socket* Specifies a socket that was created with *socket()*, has been bound to an
3486 address with *bind()*, and has issued a successful call to *listen()*.3487 *address* Either a null pointer, or a pointer to a **sockaddr** structure where the
3488 address of the connecting socket will be returned.3489 *address_len* Points to a **size_t** which on input specifies the length of the supplied
3490 **sockaddr** structure, and on output specifies the length of the stored
3491 address.3492 If *address* is not a null pointer, the address of the peer for the accepted connection is stored in the
3493 **sockaddr** structure pointed to by *address*, and the length of this address is stored in the object
3494 pointed to by *address_len*.3495 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
3496 the stored address will be truncated.3497 If the protocol permits connections by unbound clients, and the peer is not bound, then the value
3498 stored in the object pointed to by *address* is unspecified.3499 If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file
3500 descriptor for the socket, *accept()* will block until a connection is present. If the *listen()* queue is
3501 empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket,
3502 *accept()* will fail and set *errno* to [EWOULDBLOCK] or [EAGAIN].3503 The accepted socket cannot itself accept more connections. The original socket remains open
3504 and can accept more connections.3505 **RETURN VALUE**3506 Upon successful completion, *accept()* returns the nonnegative file descriptor of the accepted
3507 socket. Otherwise, -1 is returned and *errno* is set to indicate the error.3508 **ERRORS**3509 The *accept()* function will fail if:3510 [EBADF] The *socket* argument is not a valid file descriptor.

3511 [ECONNABORTED] A connection has been aborted.

3512 [ENOTSOCK] The *socket* argument does not refer to a socket.3513 [EOPNOTSUPP] The socket type of the specified socket does not support accepting
3514 connections.

3515 [EAGAIN] or [EWOULDBLOCK]

3516 O_NONBLOCK is set for the socket file descriptor and no connections are
3517 present to be accepted.

3518	[EINTR]	The <i>accept()</i> function was interrupted by a signal that was caught before a valid connection arrived.
3519		
3520	[EINVAL]	The <i>socket</i> is not accepting connections.
3521	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
3522	[ENFILE]	The maximum number of file descriptors in the system are already open.
3523		The <i>accept()</i> function may fail if:
3524	[ENOMEM]	There was insufficient memory available to complete the operation.
3525	[ENOBUFS]	No buffer space is available.
3526	[ENOSR]	There was insufficient STREAMS resources available to complete the operation.
3527		
3528	[EPROTO]	A protocol error has occurred; for example, the STREAMS protocol stack has not been initialised.
3529		

3530 APPLICATION USAGE

3531 When a connection is available, *select()* will indicate that the file descriptor for the socket is
3532 ready for reading.

3533 SEE ALSO

3534 *bind()*, *connect()*, *listen()*, *socket()*, <sys/socket>.

3535 CHANGE HISTORY

3536 First released in Issue 4.

3537 **NAME**

3538 bind — bind a name to a socket

3539 **SYNOPSIS**

3540 UX #include <sys/socket.h>

```
3541 int bind(int socket, const struct sockaddr *address,
3542         size_t address_len);
```

3543 **DESCRIPTION**

3544 The *bind()* function assigns an *address* to an unnamed socket. Sockets created with *socket()*
 3545 function are initially unnamed; they are identified only by their address family.

3546 The function takes the following arguments:

3547	<i>socket</i>	Specifies the file descriptor of the socket to be bound.
3548	<i>address</i>	Points to a sockaddr structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.
3549		
3550		
3551	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> argument.
3552		

3553 **RETURN VALUE**

3554 Upon successful completion, *bind()* returns 0. Otherwise, -1 is returned and *errno* is set to
 3555 indicate the error.

3556 **ERRORS**

3557 The *bind()* function will fail if:

3558	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
3559	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
3560	[EADDRNOTAVAIL]	The specified address is not available from the local machine.
3561	[EADDRINUSE]	The specified address is already in use.
3562	[EINVAL]	The socket is already bound to an address, and the protocol does not support binding to a new address; or the socket has been shut down.
3563		
3564	[EACCES]	The specified address is protected and the current user does not have permission to bind to it.
3565		
3566	[EAFNOSUPPORT]	The specified address is not a valid address for the address family of the specified socket.
3567		
3568	[EOPNOTSUPP]	The socket type of the specified socket does not support binding to an address.
3569		

3570 If the address family of the socket is AF_UNIX, then *bind()* will fail if:

3571	[EDESTADDRREQ] or [EISDIR]	
3572		The <i>address</i> argument is a null pointer.
3573	[EACCES]	A component of the path prefix denies search permission, or the requested name requires writing in a directory with a mode that denies write permission.
3574		
3575		
3576	[ENOTDIR]	A component of the path prefix of the pathname in <i>address</i> is not a directory.
3577		

- 3578 [ENAMETOOLONG] A component of a pathname exceeded {NAME_MAX} characters, or an
3579 entire pathname exceeded {PATH_MAX} characters.
- 3580 [ENOENT] A component of the pathname does not name an existing file or the
3581 pathname is an empty string.
- 3582 [ELOOP] Too many symbolic links were encountered in translating the pathname
3583 in *address*.
- 3584 [EIO] An I/O error occurred.
- 3585 [EROFS] The name would reside on a read-only filesystem.
- 3586 The *bind()* function may fail if:
- 3587 [EINVAL] The *address_len* argument is not a valid length for the address family.
- 3588 [EISCONN] The socket is already connected.
- 3589 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
3590 whose length exceeds {PATH_MAX}.
- 3591 [ENOBUFS] Insufficient resources were available to complete the call.
- 3592 [ENOSR] There were insufficient STREAMS resources for the operation to
3593 complete.

3594 **APPLICATION USAGE**

3595 An application program can retrieve the assigned socket name with the *getsockname()* function.

3596 **SEE ALSO**

3597 *connect()*, *getsockname()*, *listen()*, *socket()*, <sys/socket>.

3598 **CHANGE HISTORY**

3599 First released in Issue 4.

3600 **NAME**

3601 close — close a file descriptor

3602 **Note:** The **XSH** specification contains the basic definition of this interface. The following
3603 additional information pertains to Sockets.

3604 **DESCRIPTION**

3605 UX If *fdes* refers to a socket, *close()* causes the socket to be destroyed. If the socket is connection-
3606 oriented, and the `SOCK_LINGER` option is set for the socket, and the socket has untransmitted
3607 data, then *close()* will block for up to the current linger interval until all data is transmitted.

3608 **CHANGE HISTORY**

3609 First released in Issue 4.

3610 NAME

3611 connect — connect a socket

3612 SYNOPSIS

3613 ux #include <sys/socket.h>

```
3614 int connect(int socket, const struct sockaddr *address,
3615             size_t address_len);
```

3616 DESCRIPTION

3617 The *connect()* function requests a connection to be made on a socket. The function takes the
 3618 following arguments:

3619	<i>socket</i>	Specifies the file descriptor associated with the socket.
3620	<i>address</i>	Points to a sockaddr structure containing the peer address. The length and format of the address depend on the address family of the socket.
3621		
3622	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> argument.
3623		

3624 If the initiating socket is not connection-oriented, then *connect()* sets the socket's peer address,
 3625 but no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all
 3626 datagrams are sent on subsequent *send()* calls, and limits the remote sender for subsequent
 3627 *recv()* calls. If *address* is a null address for the protocol, the socket's peer address will be reset.

3628 If the initiating socket is connection-oriented, then *connect()* attempts to establish a connection
 3629 to the address specified by the *address* argument.

3630 If the connection cannot be established immediately and O_NONBLOCK is not set for the file
 3631 descriptor for the socket, *connect()* will block for up to an unspecified timeout interval until the
 3632 connection is established. If the timeout interval expires before the connection is established,
 3633 *connect()* will fail and the connection attempt will be aborted. If *connect()* is interrupted by a
 3634 signal that is caught while blocked waiting to establish a connection, *connect()* will fail and set
 3635 *errno* to [EINTR], but the connection request will not be aborted, and the connection will be
 3636 established asynchronously.

3637 If the connection cannot be established immediately and O_NONBLOCK is set for the file
 3638 descriptor for the socket, *connect()* will fail and set *errno* to [EINPROGRESS], but the connection
 3639 request will not be aborted, and the connection will be established asynchronously. Subsequent
 3640 calls to *connect()* for the same socket, before the connection is established, will fail and set *errno*
 3641 to [EALREADY].

3642 When the connection has been established asynchronously, *select()* and *poll()* will indicate that
 3643 the file descriptor for the socket is ready for writing.

3644 RETURN VALUE

3645 Upon successful completion, *connect()* returns 0. Otherwise, -1 is returned and *errno* is set to
 3646 indicate the error.

3647 ERRORS

3648 The *connect()* function will fail if:

3649	[EADDRNOTAVAIL]	The specified address is not available from the local machine.
3650	[EAFNOSUPPORT]	The specified address is not a valid address for the address family of the specified socket.
3651		
3652	[EALREADY]	A connection request is already in progress for the specified socket.

3653	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
3654	[ECONNREFUSED]	The target address was not listening for connections or refused the connection request.
3655		
3656	[EINPROGRESS]	O_NONBLOCK is set for the file descriptor for the socket and the connection cannot be immediately established; the connection will be established asynchronously.
3657		
3658		
3659	[EINTR]	The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection will be established asynchronously.
3660		
3661		
3662	[EISCONN]	The specified socket is connection-oriented and is already connected.
3663	[ENETUNREACH]	No route to the network is present.
3664	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
3665	[EPROTOTYPE]	The specified address has a different type than the socket bound to the specified peer address.
3666		
3667	[ETIMEDOUT]	The attempt to connect timed out before a connection was made.
3668		If the address family of the socket is AF_UNIX, then <i>connect()</i> will fail if:
3669	[ENOTDIR]	A component of the path prefix of the pathname in <i>address</i> is not a directory.
3670		
3671	[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
3672		
3673	[EACCES]	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
3674		
3675	[EIO]	An I/O error occurred while reading from or writing to the file system.
3676	[ELOOP]	Too many symbolic links were encountered in translating the pathname in <i>address</i> .
3677		
3678	[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
3679		
3680		The <i>connect()</i> function may fail if:
3681	[EADDRINUSE]	Attempt to establish a connection that uses addresses that are already in use.
3682		
3683	[ECONNRESET]	Remote host reset the connection request.
3684	[EHOSTUNREACH]	The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
3685		
3686	[EINVAL]	The <i>address_len</i> argument is not a valid length for the address family; or invalid address family in <i>sockaddr</i> structure.
3687		
3688	[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
3689		
3690	[ENETDOWN]	The local interface used to reach the destination is down.
3691	[ENOBUFS]	No buffer space is available.

3692 [ENOSR] There were insufficient STREAMS resources available to complete the
3693 operation.

3694 [EOPNOTSUPP] The socket is listening and can not be connected.

3695 **APPLICATION USAGE**

3696 If *connect()* fails, the state of the socket is unspecified. Portable applications should close the file
3697 descriptor and create a new socket before attempting to reconnect.

3698 **SEE ALSO**

3699 *accept(), bind(), close(), getsockname(), poll(), select(), send(), shutdown(), socket(), <sys/socket.h>*.

3700 **CHANGE HISTORY**

3701 First released in Issue 4.

3702 **NAME**

3703 fcntl — file control

3704 **Note:** The XSH specification contains the basic definition of this interface. The following
 3705 additional information pertains to Sockets.

3706 **DESCRIPTION**3707 UX The following additional values for *cmd* are defined in `<fcntl.h>`:

3708 3709 3710 3711 3712	F_GETOWN	If <i>fildev</i> refers to a socket, get the process or process group ID specified to receive SIGURG signals when out-of-band data is available. Positive values indicate a process ID; negative values, other than -1, indicate a process group ID. If <i>fildev</i> does not refer to a socket, the results are unspecified.
--------------------------------------	-----------------	--

3713 3714 3715 3716 3717	F_SETOWN	If <i>fildev</i> refers to a socket, set the process or process group ID specified to receive SIGURG signals when out-of-band data is available, using the value of the third argument, <i>arg</i> , taken as type int . Positive values indicate a process ID; negative values, other than -1, indicate a process group ID. If <i>fildev</i> does not refer to a socket, the results are unspecified.
--------------------------------------	-----------------	---

3718 **RETURN VALUE**3719 UX Upon successful completion, the value returned depends on *cmd* as follows:

3720	F_GETOWN	Value of the socket owner process or process group; this will not be -1.
------	-----------------	--

3721	F_SETOWN	Value other than -1.
------	-----------------	----------------------

3722 **CHANGE HISTORY**

3723 First released in Issue 4.

3724 **NAME**

3725 fgetpos — get current file position information

3726 **Note:** The **XSH** specification contains the basic definition of this interface. The following
3727 additional information pertains to Sockets.

3728 **ERRORS**3729 UX The *fgetpos()* function may fail if:3730 [ESPIPE] The file descriptor underlying *stream* is associated with a socket.3731 **CHANGE HISTORY**

3732 First released in Issue 4.

3733 **NAME**

3734 fsetpos — set current file position

3735 **Note:** The **XSH** specification contains the basic definition of this interface. The following
3736 additional information pertains to Sockets.

3737 **ERRORS**3738 UX The *fsetpos()* function may fail if:3739 [ESPIPE] The file descriptor underlying *stream* is associated with a socket.3740 **CHANGE HISTORY**

3741 First released in Issue 4.

3742 **NAME**

3743 ftell — return a file offset in a stream

3744 **Note:** The **XSH** specification contains the basic definition of this interface. The following
3745 additional information pertains to Sockets.

3746 **ERRORS**3747 UX The *ftell()* function may fail if:3748 [ESPIPE] The file descriptor underlying *stream* is associated with a socket.3749 **CHANGE HISTORY**

3750 First released in Issue 4.

3751 **NAME**

3752 getpeername — get the name of the peer socket

3753 **SYNOPSIS**3754 ux

```
#include <sys/socket.h>
```

3755

```
int getpeername(int socket, struct sockaddr *address,
```


3756

```
size_t *address_len);
```

3757 **DESCRIPTION**3758 The *getpeername()* function retrieves the peer address of the specified socket, stores this address
3759 in the **sockaddr** structure pointed to by the *address* argument, and stores the length of this
3760 address in the object pointed to by the *address_len* argument.3761 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
3762 the stored address will be truncated.3763 If the protocol permits connections by unbound clients, and the peer is not bound, then the value
3764 stored in the object pointed to by *address* is unspecified.3765 **RETURN VALUE**3766 Upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate
3767 the error.3768 **ERRORS**3769 The *getpeername()* function will fail if:3770 [EBADF] The *socket* argument is not a valid file descriptor.3771 [ENOTSOCK] The *socket* argument does not refer to a socket.3772 [ENOTCONN] The socket is not connected or otherwise has not had the peer
3773 prespecified.

3774 [EINVAL] The socket has been shut down.

3775 [EOPNOTSUPP] The operation is not supported for the socket protocol.

3776 The *getpeername()* function may fail if:

3777 [ENOBUFS] Insufficient resources were available in the system to complete the call.

3778 [ENOSR] There were insufficient STREAMS resources available for the operation to
3779 complete.3780 **SEE ALSO**3781 *accept()*, *bind()*, *getsockname()*, *socket()*, *<sys/socket.h>*.3782 **CHANGE HISTORY**

3783 First released in Issue 4.

3784 **NAME**

3785 getsockname — get the socket name

3786 **SYNOPSIS**3787 **UX** #include <sys/socket.h>3788 int getsockname(int *socket*, struct sockaddr **address*,
3789 size_t **address_len*);3790 **DESCRIPTION**3791 The *getsockname()* function retrieves the locally-bound name of the specified socket, stores this
3792 address in the **sockaddr** structure pointed to by the *address* argument, and stores the length of
3793 this address in the object pointed to by the *address_len* argument.3794 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
3795 the stored address will be truncated.3796 If the socket has not been bound to a local name, the value stored in the object pointed to by
3797 *address* is unspecified.3798 **RETURN VALUE**3799 Upon successful completion, 0 is returned, the *address* argument points to the address of the
3800 socket, and the *address_len* argument points to the length of the address. Otherwise, -1 is
3801 returned and *errno* is set to indicate the error.3802 **ERRORS**3803 The *getsockname()* function will fail:3804 [EBADF] The *socket* argument is not a valid file descriptor.3805 [ENOTSOCK] The *socket* argument does not refer to a socket.

3806 [EOPNOTSUPP] The operation is not supported for this socket's protocol.

3807 The *getsockname()* function may fail if:

3808 [EINVAL] The socket has been shut down.

3809 [ENOBUFS] Insufficient resources were available in the system to complete the call.

3810 [ENOSR] There were insufficient STREAMS resources available for the operation to
3811 complete.3812 **SEE ALSO**3813 *accept()*, *bind()*, *getpeername()*, *socket()*, <sys/socket.h>.3814 **CHANGE HISTORY**

3815 First released in Issue 4.

3816 **NAME**

3817 getsockopt — get the socket options

3818 **SYNOPSIS**3819 ux

```
#include <sys/socket.h>
```

3820

```
int getsockopt(int socket, int level, int option_name, void *option_value,  
3821 size_t *option_len);
```

3822 **DESCRIPTION**3823 The *getsockopt()* function retrieves the value for the option specified by the *option_name*
3824 argument for the socket specified by the *socket* argument. If the size of the option value is
3825 greater than *option_len*, the value stored in the object pointed to by the *option_value* argument
3826 will be silently truncated. Otherwise, the object pointed to by the *option_len* argument will be
3827 modified to indicate the actual length of the value.3828 The *level* argument specifies the protocol level at which the option resides. To retrieve options at
3829 the socket level, specify the *level* argument as SOL_SOCKET. To retrieve options at other levels,
3830 supply the appropriate protocol number for the protocol controlling the option. For example, to
3831 indicate that an option will be interpreted by the TCP (Transport Control Protocol), set *level* to
3832 the protocol number of TCP, as defined in the `<netinet/in.h>` header, or as determined by using
3833 *getprotobyname()* function.3834 The *option_name* argument specifies a single option to be retrieved. It can be one of the following
3835 values defined in `<sys/socket.h>`:3836 SO_DEBUG Reports whether debugging information is being recorded. This option
3837 stores an **int** value.3838 SO_ACCEPTCONN Reports whether socket listening is enabled. This option stores an **int**
3839 value.3840 SO_BROADCAST Reports whether transmission of broadcast messages is supported, if this
3841 is supported by the protocol. This option stores an **int** value.3842 SO_REUSEADDR Reports whether the rules used in validating addresses supplied to *bind()*
3843 should allow reuse of local addresses, if this is supported by the protocol.
3844 This option stores an **int** value.3845 SO_KEEPALIVE Reports whether connections are kept active with periodic transmission
3846 of messages, if this is supported by the protocol.3847 If the connected socket fails to respond to these messages, the connection
3848 is broken and processes writing to that socket are notified with a SIGPIPE
3849 signal. This option stores an **int** value.3850 SO_LINGER Reports whether the socket lingers on *close()* if data is present. If
3851 SO_LINGER is set, the system blocks the process during *close()* until it
3852 can transmit the data or until the end of the interval indicated by the
3853 **linger** member, whichever comes first. If SO_LINGER is not specified,
3854 and *close()* is issued, the system handles the call in a way that allows the
3855 process to continue as quickly as possible. This option stores a **linger**
3856 structure.3857 SO_OOINLINE Reports whether the socket leaves received out-of-band data (data
3858 marked urgent) in line. This option stores an **int** value.3859 SO_SNDBUF Reports send buffer size information. This option stores an **int** value.

3860 SO_RCVBUF Reports receive buffer size information. This option stores an **int** value.
3861 SO_ERROR Reports information about error status and clears it. This option stores an
3862 **int** value.
3863 SO_TYPE Reports the socket type. This option stores an **int** value.
3864 For boolean options, 0 indicates that the option is disabled and 1 indicates that the option is
3865 enabled.
3866 Options at other protocol levels vary in format and name.

3867 RETURN VALUE

3868 Upon successful completion, *getsockopt()* returns 0. Otherwise, -1 is returned and *errno* is set to
3869 indicate the error.

3870 ERRORS

3871 The *getsockopt()* function will fail if:

3872 [EBADF] The *socket* argument is not a valid file descriptor.
3873 [ENOPROTOOPT] The option is not supported by the protocol.
3874 [ENOTSOCK] The *socket* argument does not refer to a socket.
3875 [EINVAL] The specified option is invalid at the specified socket level.
3876 [EOPNOTSUPP] The operation is not supported by the socket protocol.

3877 The *getsockopt()* function may fail if:

3878 [EINVAL] The socket has been shut down.
3879 [ENOBUFS] Insufficient resources are available in the system to complete the call.
3880 [ENOSR] There were insufficient STREAMS resources available for the operation to
3881 complete.

3882 SEE ALSO

3883 *bind()*, *close()*, *endprotoent()*, *setsockopt()*, *socket()*, <sys/socket.h>.

3884 CHANGE HISTORY

3885 First released in Issue 4.

3886 **NAME**

3887 listen — listen for socket connections and limit the queue of incoming connections

3888 **SYNOPSIS**

```
3889 ux #include <sys/socket.h>
```

```
3890 int listen(int socket, int backlog);
```

3891 **DESCRIPTION**

3892 The *listen()* function marks a connection-oriented socket, specified by the *socket* argument, as
3893 accepting connections, and limits the number of outstanding connections in the socket's listen
3894 queue to the value specified by the *backlog* argument.

3895 If *listen()* is called with a *backlog* argument value that is less than 0, the function sets the length
3896 of the socket's listen queue to 0.

3897 Implementations may limit the length of the socket's listen queue. If *backlog* exceeds the
3898 implementation-dependent maximum queue length, the length of the socket's listen queue will
3899 be set to the maximum supported value.

3900 **RETURN VALUE**

3901 Upon successful completions, *listen()* returns 0. Otherwise, -1 is returned and *errno* is set to
3902 indicate the error.

3903 **ERRORS**

3904 The *listen()* function will fail if:

3905 [EBADF] The *socket* argument is not a valid file descriptor.

3906 [ENOTSOCK] The *socket* argument does not refer to a socket.

3907 [EOPNOTSUPP] The socket protocol does not support *listen()*.

3908 [EINVAL] The *socket* is already connected.

3909 [EDESTADDRREQ] The socket is not bound to a local address, and the protocol does not
3910 support listening on an unbound socket.

3911 The *listen()* function may fail if:

3912 [EINVAL] The *socket* has been shut down.

3913 [ENOBUFS] Insufficient resources are available in the system to complete the call.

3914 **SEE ALSO**

3915 *accept()*, *connect()*, *socket()*, <sys/socket.h>.

3916 **CHANGE HISTORY**

3917 First released in Issue 4.

3918 **NAME**

3919 lseek — move read/write file offset

3920 **Note:** The **XSH** specification contains the basic definition of this interface. The following
3921 additional information pertains to Sockets.

3922 **ERRORS**3923 UX The *lseek()* function will fail if:3924 [ESPIPE] The file descriptor underlying *stream* is associated with a socket.3925 **CHANGE HISTORY**

3926 First released in Issue 4.

3927 **NAME**

3928 poll — input/output multiplexing

3929 **Note:** The **XSH** specification contains the basic definition of this interface. The following
3930 additional information pertains to Sockets.

3931 **DESCRIPTION**3932 UX The *poll()* function supports sockets.

3933 A file descriptor for a socket that is listening for connections will indicate that it is ready for
3934 reading, once connections are available. A file descriptor for a socket that is connecting
3935 asynchronously will indicate that it is ready for writing, once a connection has been established.

3936 **CHANGE HISTORY**

3937 First released in Issue 4.

3938 **NAME**

3939 read, readv — read from file

3940 **Note:** The **XSH** specification contains the basic definition of this interface. The following
3941 additional information pertains to Sockets.

3942 **DESCRIPTION**3943 UX If *fildev* refers to a socket, *read()* is equivalent to *recv()* with no flags set.3944 **CHANGE HISTORY**

3945 First released in Issue 4.

3946 **NAME**

3947 recv — receive a message from a connected socket

3948 **SYNOPSIS**

3949 ux #include <sys/socket.h>

3950 ssize_t recv(int *socket*, void **buffer*, size_t *length*, int *flags*);3951 **DESCRIPTION**3952 The *recv()* function receives messages from a connected socket. The function takes the following
3953 arguments:

3954	<i>socket</i>	Specifies the socket file descriptor.
3955	<i>buffer</i>	Points to a buffer where the message should be stored.
3956	<i>length</i>	Specifies the length in bytes of the buffer pointed to by the <i>buffer</i> 3957 argument.
3958	<i>flags</i>	Specifies the type of message reception. Values of this argument are 3959 formed by logically OR'ing zero or more of the following values:
3960	MSG_PEEK	Peeks at an incoming message. The data is treated 3961 as unread and the next <i>recv()</i> or similar function 3962 will still return this data.
3963	MSG_OOB	Requests out-of-band data. The significance and 3964 semantics of out-of-band data are protocol- 3965 specific.
3966	MSG_WAITALL	Requests that the function block until the full 3967 amount of data requested can be returned. The 3968 function may return a smaller amount of data if a 3969 signal is caught, the connection is terminated, or 3970 an error is pending for the socket.

3971 The *recv()* function returns the length of the message written to the buffer pointed to by the
3972 *buffer* argument. For message-based sockets such as SOCK_DGRAM and SOCK_SEQPACKET,
3973 the entire message must be read in a single operation. If a message is too long to fit in the
3974 supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes are discarded.
3975 For stream-based sockets such as SOCK_STREAM, message boundaries are ignored. In this
3976 case, data is returned to the user as soon as it becomes available, and no data is discarded.

3977 If the MSG_WAITALL flag is not set, data will be returned only up to the end of the first
3978 message.

3979 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
3980 descriptor, *recv()* blocks until a message arrives. If no messages are available at the socket and
3981 O_NONBLOCK is set on the socket's file descriptor, *recv()* fails and sets *errno* to
3982 [EWOULDBLOCK] or [EAGAIN].

3983 **RETURN VALUE**

3984 Upon successful completion, *recv()* returns the length of the message in bytes. If no messages
3985 are available to be received and the peer has performed an orderly shutdown, *recv()* returns 0.
3986 Otherwise, -1 is returned and *errno* is set to indicate the error.

3987 **ERRORS**3988 The *recv()* function will fail if:

- 3989 [EBADF] The *socket* argument is not a valid file descriptor.
- 3990 [ECONNRESET] A connection was forcibly closed by a peer.
- 3991 [EINTR] The *recv()* function was interrupted by a signal that was caught, before
3992 any data was available.
- 3993 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.
- 3994 [ENOTCONN] A receive is attempted on a connection-oriented socket that is not
3995 connected.
- 3996 [ENOTSOCK] The *socket* argument does not refer to a socket.
- 3997 [EOPNOTSUPP] The specified flags are not supported for this socket type or protocol.
- 3998 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
3999 transmission timeout on active connection.
- 4000 [EWOULDBLOCK] or [EAGAIN]
4001 The socket's file descriptor is marked O_NONBLOCK and no data is
4002 waiting to be received; or MSG_OOB is set and no out-of-band data is
4003 available and either the socket's file descriptor is marked O_NONBLOCK
4004 or the socket does not support blocking to await out-of-band data.

4005 The *recv()* function may fail if:

- 4006 [EIO] An I/O error occurred while reading from or writing to the file system.
- 4007 [ENOBUFS] Insufficient resources were available in the system to perform the
4008 operation.
- 4009 [ENOMEM] Insufficient memory was available to fulfill the request.
- 4010 [ENOSR] There were insufficient STREAMS resources available for the operation to
4011 complete.

4012 **APPLICATION USAGE**4013 The *recv()* function is identical to *recvfrom()* with a zero *address_len* argument, and to *read()* if no
4014 flags are used.4015 The *select()* and *poll()* functions can be used to determine when data is available to be received.4016 **SEE ALSO**4017 *poll()*, *read()*, *recvmsg()*, *recvfrom()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*,
4018 *write()*, <*sys/socket.h*>.4019 **CHANGE HISTORY**

4020 First released in Issue 4.

4021 **NAME**

4022 recvfrom — receive a message from a socket

4023 **SYNOPSIS**4024 **UX** #include <sys/socket.h>4025 ssize_t recvfrom(int *socket*, void **buffer*, size_t *length*, int *flags*,
4026 struct sockaddr **address*, size_t **address_len*);4027 **DESCRIPTION**4028 The *recvfrom()* function receives a message from a connection-oriented or connectionless socket.
4029 It is normally used with connectionless sockets because it permits the application to retrieve the
4030 source address of received data.

4031 The function takes the following arguments:

4032	<i>socket</i>	Specifies the socket file descriptor.
4033	<i>buffer</i>	Points to the buffer where the message should be stored.
4034	<i>length</i>	Specifies the length in bytes of the buffer pointed to by the <i>buffer</i> 4035 argument.
4036	<i>flags</i>	Specifies the type of message reception. Values of this argument are 4037 formed by logically OR'ing zero or more of the following values:
4038	MSG_PEEK	Peeks at an incoming message. The data is treated 4039 as unread and the next <i>recvfrom()</i> or similar 4040 function will still return this data.
4041	MSG_OOB	Requests out-of-band data. The significance and 4042 semantics of out-of-band data are protocol- 4043 specific.
4044	MSG_WAITALL	Requests that the function block until the full 4045 amount of data requested can be returned. The 4046 function may return a smaller amount of data if a 4047 signal is caught, the connection is terminated, or 4048 an error is pending for the socket.
4049	<i>address</i>	A null pointer, or points to a sockaddr structure in which the sending 4050 address is to be stored. The length and format of the address depend on 4051 the address family of the socket.
4052	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> 4053 argument.

4054 The *recvfrom()* function returns the length of the message written to the buffer pointed to by the
4055 *buffer* argument. For message-based sockets such as SOCK_DGRAM and SOCK_SEQPACKET,
4056 the entire message must be read in a single operation. If a message is too long to fit in the
4057 supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes are discarded.
4058 For stream-based sockets such as SOCK_STREAM, message boundaries are ignored. In this
4059 case, data is returned to the user as soon as it becomes available, and no data is discarded.

4060 If the MSG_WAITALL flag is not set, data will be returned only up to the end of the first
4061 message.

4062 Not all protocols provide the source address for messages. If the *address* argument is not a null
4063 pointer and the protocol provides the source address of messages, the source address of the
4064 received message is stored in the **sockaddr** structure pointed to by the *address* argument, and the

- 4065 length of this address is stored in the object pointed to by the *address_len* argument.
- 4066 If the actual length of the address is greater than the length of the supplied **sockaddr** structure,
4067 the stored address will be truncated.
- 4068 If the *address* argument is not a null pointer and the protocol does not provide the source address
4069 of messages, the the value stored in the object pointed to by *address* is unspecified.
- 4070 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
4071 descriptor, *recvfrom()* blocks until a message arrives. If no messages are available at the socket
4072 and O_NONBLOCK is set on the socket's file descriptor, *recvfrom()* fails and sets *errno* to
4073 [EWOULDBLOCK] or [EAGAIN].
- 4074 **RETURN VALUE**
- 4075 Upon successful completion, *recvfrom()* returns the length of the message in bytes. If no
4076 messages are available to be received and the peer has performed an orderly shutdown,
4077 *recvfrom()* returns 0. Otherwise the function returns -1 and sets *errno* to indicate the error.
- 4078 **ERRORS**
- 4079 The *recvfrom()* function will fail if:
- 4080 [EBADF] The *socket* argument is not a valid file descriptor.
- 4081 [ECONNRESET] A connection was forcibly closed by a peer.
- 4082 [EINTR] A signal interrupted *recvfrom()* before any data was available.
- 4083 [EINVAL] The MSG_OOB flag is set and no out-of-band data is available.
- 4084 [ENOTCONN] A receive is attempted on a connection-oriented socket that is not
4085 connected.
- 4086 [ENOTSOCK] The *socket* argument does not refer to a socket.
- 4087 [EOPNOTSUPP] The specified flags are not supported for this socket type.
- 4088 [ETIMEDOUT] The connection timed out during connection establishment, or due to a
4089 transmission timeout on active connection.
- 4090 [EWOULDBLOCK] or [EAGAIN]
- 4091 The socket's file descriptor is marked O_NONBLOCK and no data is
4092 waiting to be received; or MSG_OOB is set and no out-of-band data is
4093 available and either the socket's file descriptor is marked O_NONBLOCK
4094 or the socket does not support blocking to await out-of-band data.
- 4095 The *recvfrom()* function may fail if:
- 4096 [EIO] An I/O error occurred while reading from or writing to the file system.
- 4097 [ENOBUFS] Insufficient resources were available in the system to perform the
4098 operation.
- 4099 [ENOMEM] Insufficient memory was available to fulfill the request.
- 4100 [ENOSR] There were insufficient STREAMS resources available for the operation to
4101 complete.
- 4102 **APPLICATION USAGE**
- 4103 The *select()* and *poll()* functions can be used to determine when data is available to be received.

4104 **SEE ALSO**

4105 *poll()*, *read()*, *recv()*, *recvmsg()*, *select()* *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*, *write()*,
4106 **<sys/socket.h>**.

4107 **CHANGE HISTORY**

4108 First released in Issue 4.

4109 NAME

4110 recvmsg — receive a message from a socket

4111 SYNOPSIS

4112 ux `#include <sys/socket.h>`4113 `ssize_t recvmsg(int socket, struct msghdr *message, int flags);`

4114 DESCRIPTION

4115 The *recvmsg()* function receives a message from a connection-oriented or connectionless socket.
 4116 It is normally used with connectionless sockets because it permits the application to retrieve the
 4117 source address of received data.

4118 The function takes the following arguments:

4119	<i>socket</i>	Specifies the socket file descriptor.
4120	<i>message</i>	Points to a msghdr structure, containing both the buffer to store the source address and the buffers for the incoming message. The length and format of the address depend on the address family of the socket. The msg_flags member is ignored on input, but may contain meaningful values on output.
4125	<i>flags</i>	Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:
4127	MSG_OOB	Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
4130	MSG_PEEK	Peeks at the incoming message.
4131	MSG_WAITALL	Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, the connection is terminated, or an error is pending for the socket.

4136 The *recvmsg()* function receives messages from unconnected or connected sockets and returns
 4137 the length of the message.

4138 The *recvmsg()* function returns the total length of the message. For message-based sockets such
 4139 as SOCK_DGRAM and SOCK_SEQPACKET, the entire message must be read in a single
 4140 operation. If a message is too long to fit in the supplied buffers, and MSG_PEEK is not set in the
 4141 *flags* argument, the excess bytes are discarded, and MSG_TRUNC is set in the **msg_flags**
 4142 member of the **msghdr** structure. For stream-based sockets such as SOCK_STREAM, message
 4143 boundaries are ignored. In this case, data is returned to the user as soon as it becomes available,
 4144 and no data is discarded.

4145 If the MSG_WAITALL flag is not set, data will be returned only up to the end of the first
 4146 message.

4147 If no messages are available at the socket and O_NONBLOCK is not set on the socket's file
 4148 descriptor, *recvfrom()* blocks until a message arrives. If no messages are available at the socket
 4149 and O_NONBLOCK is set on the socket's file descriptor, *recvfrom()* function fails and sets *errno*
 4150 to [EWOULDBLOCK] or [EAGAIN].

4151 In the **msghdr** structure, the **msg_name** and **msg_namelen** members specify the source address
 4152 if the socket is unconnected. If the socket is connected, the **msg_name** and **msg_namelen**
 4153 members are ignored. The **msg_name** member may be a null pointer if no names are desired or

4154 required. The **msg_iov** and **msg_iovlen** members describe the scatter/gather locations.
 4155 On successful completion, the **msg_flags** member of the message header is the bitwise-inclusive
 4156 OR of all of the following flags that indicate conditions detected for the received message:.

4157	MSG_EOR	End of record was received (if supported by the protocol).
4158	MSG_OOB	Out-of-band data was received.
4159	MSG_TRUNC	Normal data was truncated.
4160	MSG_CTRUNC	Control data was truncated.

4161 RETURN VALUE

4162 Upon successful completion, *recvmsg()* returns the length of the message in bytes. If no
 4163 messages are available to be received and the peer has performed an orderly shutdown,
 4164 *recvmsg()* returns 0. Otherwise, -1 is returned and *errno* is set to indicate the error.

4165 ERRORS

4166 The *recvmsg()* function will fail if:

4167	[EBADF]	The <i>socket</i> argument is not a valid open file descriptor.
4168	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
4169	[EINVAL]	The sum of the iov_len values overflows an ssize_t .
4170	[EWOULDBLOCK] or [EAGAIN]	The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.
4171		
4172		
4173		
4174		
4175	[EINTR]	This function was interrupted by a signal before any data was available.
4176	[EOPNOTSUPP]	The specified flags are not supported for this socket type.
4177	[ENOTCONN]	A receive is attempted on a connection-oriented socket that is not connected.
4178		
4179	[ETIMEDOUT]	The connection timed out during connection establishment, or due to a transmission timeout on active connection.
4180		
4181	[EINVAL]	The MSG_OOB flag is set and no out-of-band data is available.
4182	[ECONNRESET]	A connection was forcibly closed by a peer.

4183 The *recvmsg()* function may fail if:

4184	[EINVAL]	The msg_iovlen member of the msghdr structure pointed to by <i>msg</i> is less than or equal to 0, or is greater than {IOV_MAX}.
4185		
4186	[EIO]	An IO error occurred while reading from or writing to the file system.
4187	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
4188		
4189	[ENOMEM]	Insufficient memory was available to fulfill the request.
4190	[ENOSR]	There were insufficient STREAMS resources available for the operation to complete.
4191		

4192 APPLICATION USAGE

4193 The *select()* and *poll()* functions can be used to determine when data is available to be received.

4194 **SEE ALSO**

4195 *poll()*, *recv()*, *recvfrom()*, *select()*, *send()*, *sendmsg()*, *sendto()*, *shutdown()*, *socket()*,
4196 *<sys/socket.h>*.

4197 **CHANGE HISTORY**

4198 First released in Issue 4.

4199 **NAME**

4200 select — synchronous I/O multiplexing

4201 **Note:** The **XSH** specification contains the basic definition of this interface. The following
4202 additional information pertains to Sockets.4203 **DESCRIPTION**4204 UX A file descriptor for a socket that is listening for connections will indicate that it is ready for
4205 reading, when connections are available. A file descriptor for a socket that is connecting
4206 asynchronously will indicate that it is ready for writing, when a connection has been established.
42074208 **CHANGE HISTORY**

4209 First released in Issue 4.

4210 **NAME**4211 `send` — send a message on a socket4212 **SYNOPSIS**4213 `UX` `#include <sys/socket.h>`4214 `ssize_t send(int socket, const void *buffer, size_t length, int flags);`4215 **DESCRIPTION**

4216	<i>socket</i>	Specifies the socket file descriptor.
4217	<i>buffer</i>	Points to the buffer containing the message to send.
4218	<i>length</i>	Specifies the length of the message in bytes.
4219	<i>flags</i>	Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:
4221	MSG_EOR	Terminates a record (if supported by the protocol)
4222	MSG_OOB	Sends out-of-band data on sockets that support out-of-band communications. The significance and semantics of out-of-band data are protocol-specific.
4223		
4224		
4225		

4226 The `send()` function initiates transmission of a message from the specified socket to its peer. The
 4227 `send()` function sends a message only when the socket is connected.

4228 The length of the message to be sent is specified by the *length* argument. If the message is too
 4229 long to pass through the underlying protocol, `send()` fails and no data is transmitted.

4230 Successful completion of a call to `send()` does not guarantee delivery of the message. A return
 4231 value of `-1` indicates only locally-detected errors.

4232 If space is not available at the sending socket to hold the message to be transmitted and the
 4233 socket file descriptor does not have `O_NONBLOCK` set, `send()` blocks until space is available. If
 4234 space is not available at the sending socket to hold the message to be transmitted and the socket
 4235 file descriptor does have `O_NONBLOCK` set, `send()` will fail. The `select()` and `poll()` functions
 4236 can be used to determine when it is possible to send more data.

4237 **RETURN VALUE**

4238 Upon successful completion, `send()` returns the number of bytes sent. Otherwise, `-1` is returned
 4239 and *errno* is set to indicate the error.

4240 **APPLICATION USAGE**

4241 The `send()` function is identical to `sendto()` with a null pointer *dest_len* argument, and to `write()` if
 4242 no flags are used.

4243 **ERRORS**

4244 The `send()` function will fail if:

4245	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
4246	[ECONNRESET]	A connection was forcibly closed by a peer.
4247	[EDESTADDRREQ]	The socket is not connection-oriented and no peer address is set.
4248	[EINTR]	A signal interrupted <code>send()</code> before any data was transmitted.
4249	[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.

- 4250 [ENOTCONN] The socket is not connected or otherwise has not had the peer
4251 prespecified.
- 4252 [ENOTSOCK] The *socket* argument does not refer to a socket.
- 4253 [EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one
4254 or more of the values set in *flags*.
- 4255 [EPIPE] The socket is shut down for writing, or the socket is connection-oriented
4256 and the peer is closed or shut down for reading. In the latter case, and if
4257 the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to
4258 the calling process.
- 4259 [EWOULDBLOCK] or [EAGAIN]
4260 The socket's file descriptor is marked O_NONBLOCK and the requested
4261 operation would block.
- 4262 The *send()* function may fail if:
- 4263 [ENETDOWN] The local interface used to reach the destination is down.
- 4264 [ENETUNREACH] No route to the network is present.
- 4265 [ENOBUFS] Insufficient resources were available in the system to perform the
4266 operation.
- 4267 [ENOSR] There were insufficient STREAMS resources available for the operation to
4268 complete.
- 4269 [EIO] An I/O error occurred while reading from or writing to the file system.
- 4270 **SEE ALSO**
4271 *connect()*, *getsockopt()*, *poll()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *sendmsg()*, *sendto()*,
4272 *setsockopt()*, *shutdown()*, *socket()*, <sys/socket.h>.
- 4273 **CHANGE HISTORY**
4274 First released in Issue 4.

4275 **NAME**

4276 sendmsg — send a message on a socket using a message structure

4277 **SYNOPSIS**

4278 UX #include <sys/socket.h>

4279 ssize_t sendmsg (int *socket*, const struct msghdr **message*, int *flags*);4280 **DESCRIPTION**

4281 The *sendmsg()* function sends a message through a connection-oriented or connectionless socket.
 4282 If the socket is connectionless, the message will be sent to the address specified by *msghdr*. If the
 4283 socket is connection-oriented, the destination address in *msghdr* is ignored.

4284 The function takes the following arguments:

4285	<i>socket</i>	Specifies the socket file descriptor.
4286	<i>message</i>	Points to a msghdr structure, containing both the destination address and the buffers for the outgoing message. The length and format of the address depend on the address family of the socket. The msg_flags member is ignored.
4287		
4288		
4289		
4290	<i>flags</i>	Specifies the type of message transmission. The application may specify 0 or the following flag:
4291		
4292	MSG_EOR	Terminates a record (if supported by the protocol)
4293	MSG_OOB	Sends out-of-band data on sockets that support out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
4294		
4295		

4296 Successful completion of a call to *sendmsg()* does not guarantee delivery of the message. A
 4297 return value of -1 indicates only locally-detected errors.

4298 If space is not available at the sending socket to hold the message to be transmitted and the
 4299 socket file descriptor does not have O_NONBLOCK set, *sendmsg()* function blocks until space is
 4300 available. If space is not available at the sending socket to hold the message to be transmitted
 4301 and the socket file descriptor does have O_NONBLOCK set, *sendmsg()* function will fail.

4302 If the socket protocol supports broadcast and the specified address is a broadcast address for the
 4303 socket protocol, *sendmsg()* will fail if the SO_BROADCAST option is not set for the socket.

4304 **RETURN VALUE**

4305 Upon successful completion, *sendmsg()* function returns the number of bytes sent. Otherwise,
 4306 -1 is returned and *errno* is set to indicate the error.

4307 **ERRORS**4308 The *sendmsg()* function will fail if:

4309	[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
4310	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
4311	[ECONNRESET]	A connection was forcibly closed by a peer.
4312	[EINTR]	A signal interrupted <i>sendmsg()</i> before any data was transmitted.
4313	[EINVAL]	The sum of the iov_len values overflows an ssize_t .
4314	[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.
4315	[ENOTCONN]	The socket is connection-oriented but is not connected.

4316	[ENOTSOCK]	The <i>socket</i> argument does not refer a socket.
4317	[EOPNOTSUPP]	The <i>socket</i> argument is associated with a socket that does not support one or more of the values set in <i>flags</i> .
4318		
4319	[EPIPE]	The socket is shut down for writing, or the socket is connection-oriented and the peer is closed or shut down for reading. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.
4320		
4321		
4322		
4323	[EWOULDBLOCK] or [EAGAIN]	The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.
4324		
4325		
4326		If the address family of the socket is AF_UNIX, then <i>sendmsg()</i> will fail if:
4327	[EACCES]	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
4328		
4329	[EIO]	An I/O error occurred while reading from or writing to the file system.
4330	[ELOOP]	Too many symbolic links were encountered in translating the pathname in the socket address.
4331		
4332	[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
4333		
4334	[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
4335		
4336	[ENOTDIR]	A component of the path prefix of the pathname in the socket address is not a directory.
4337		
4338		The <i>sendmsg()</i> function may fail if:
4339	[EDESTADDRREQ]	The socket is not connection-oriented and does not have its peer address set, and no destination address was specified.
4340		
4341	[EHOSTUNREACH]	The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
4342		
4343	[EINVAL]	The <i>msg_iovlen</i> member of the <i>msg_hdr</i> structure pointed to by <i>msg</i> is less than or equal to 0, or is greater than {IOV_MAX}.
4344		
4345	[EIO]	An I/O error occurred while reading from or writing to the file system.
4346	[EISCONN]	A destination address was specified and the socket is connection-oriented and is already connected.
4347		
4348	[ENETDOWN]	The local interface used to reach the destination is down.
4349	[ENETUNREACH]	No route to the network is present.
4350	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
4351		
4352	[ENOMEM]	Insufficient memory was available to fulfill the request.
4353	[ENOSR]	There were insufficient STREAMS resources available for the operation to complete.
4354		

- 4355 If the address family of the socket is AF_UNIX, then *sendmsg()* may fail if:
- 4356 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
4357 whose length exceeds {PATH_MAX}.
- 4358 **APPLICATION USAGE**
- 4359 The *select()* and *poll()* functions can be used to determine when it is possible to send more data.
- 4360 **SEE ALSO**
- 4361 *getsockopt()*, *poll()* *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *send()*, *sendto()*, *setsockopt()*, *shutdown()*,
4362 *socket()*, <sys/socket.h>.
- 4363 **CHANGE HISTORY**
- 4364 First released in Issue 4.

4365 **NAME**4366 `sendto` — send a message on a socket4367 **SYNOPSIS**4368 `UX` `#include <sys/socket.h>`

```
4369 ssize_t sendto(int socket, const void *message, size_t length, int flags,
4370                const struct sockaddr *dest_addr, size_t dest_len);
```

4371 **DESCRIPTION**

4372 The `sendto()` function sends a message through a connection-oriented or connectionless socket.
 4373 If the socket is connectionless, the message will be sent to the address specified by `dest_addr`. If
 4374 the socket is connection-oriented, `dest_addr` is ignored.

4375 The function takes the following arguments:

4376	<code>socket</code>	Specifies the socket file descriptor.
4377	<code>message</code>	Points to a buffer containing the message to be sent.
4378	<code>length</code>	Specifies the size of the message in bytes.
4379	<code>flags</code>	Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:
4381	<code>MSG_EOR</code>	Terminates a record (if supported by the protocol)
4382	<code>MSG_OOB</code>	Sends out-of-band data on sockets that support out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
4383		
4384		
4385	<code>dest_addr</code>	Points to a sockaddr structure containing the destination address. The length and format of the address depend on the address family of the socket.
4386		
4387		
4388	<code>dest_len</code>	Specifies the length of the sockaddr structure pointed to by the <code>dest_addr</code> argument.
4389		

4390 If the socket protocol supports broadcast and the specified address is a broadcast address for the
 4391 socket protocol, `sendto()` will fail if the `SO_BROADCAST` option is not set for the socket.

4392 The `dest_addr` argument specifies the address of the target. The `length` argument specifies the
 4393 length of the message.

4394 Successful completion of a call to `sendto()` does not guarantee delivery of the message. A return
 4395 value of `-1` indicates only locally-detected errors.

4396 If space is not available at the sending socket to hold the message to be transmitted and the
 4397 socket file descriptor does not have `O_NONBLOCK` set, `sendto()` blocks until space is available.
 4398 If space is not available at the sending socket to hold the message to be transmitted and the
 4399 socket file descriptor does have `O_NONBLOCK` set, `sendto()` will fail.

4400 **RETURN VALUE**

4401 Upon successful completion, `sendto()` returns the number of bytes sent. Otherwise, `-1` is
 4402 returned and `errno` is set to indicate the error.

4403 **ERRORS**

4404 The `sendto()` function will fail if:

4405 `[EAFNOSUPPORT]` Addresses in the specified address family cannot be used with this socket.

4406	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
4407	[ECONNRESET]	A connection was forcibly closed by a peer.
4408	[EINTR]	A signal interrupted <i>sendto()</i> before any data was transmitted.
4409	[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.
4410	[ENOTCONN]	The socket is connection-oriented but is not connected.
4411	[ENOTSOCK]	The <i>socket</i> argument does not refer to a socket.
4412	[EOPNOTSUPP]	The <i>socket</i> argument is associated with a socket that does not support one or more of the values set in <i>flags</i> .
4414	[EPIPE]	The socket is shut down for writing, or the socket is connection-oriented and the peer is closed or shut down for reading. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.
4415		
4416		
4417		
4418	[EWOULDBLOCK] or [EAGAIN]	
4419		The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.
4420		
4421		If the address family of the socket is AF_UNIX, then <i>sendto()</i> will fail if:
4422	[EACCES]	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
4423		
4424	[EIO]	An I/O error occurred while reading from or writing to the file system.
4425	[ELOOP]	Too many symbolic links were encountered in translating the pathname in the socket address.
4426		
4427	[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
4428		
4429	[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
4430		
4431	[ENOTDIR]	A component of the path prefix of the pathname in the socket address is not a directory.
4432		
4433		The <i>sendto()</i> function may fail if:
4434	[EDESTADDRREQ]	The socket is not connection-oriented and does not have its peer address set, and no destination address was specified.
4435		
4436	[EHOSTUNREACH]	The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
4437		
4438	[EINVAL]	The <i>dest_len</i> argument is not a valid length for the address family.
4439	[EIO]	An I/O error occurred while reading from or writing to the file system.
4440	[EISCONN]	A destination address was specified and the socket is connection-oriented and is already connected.
4441		
4442	[ENETDOWN]	The local interface used to reach the destination is down.
4443	[ENETUNREACH]	No route to the network is present.
4444	[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
4445		

- 4446 [ENOMEM] Insufficient memory was available to fulfill the request.
- 4447 [ENOSR] There were insufficient STREAMS resources available for the operation to
4448 complete.
- 4449 If the address family of the socket is AF_UNIX, then *sendto()* may fail if:
- 4450 [ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result
4451 whose length exceeds {PATH_MAX}.
- 4452 **APPLICATION USAGE**
- 4453 The *select()* and *poll()* functions can be used to determine when it is possible to send more data.
- 4454 **SEE ALSO**
- 4455 *getsockopt()*, *poll()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *send()*, *sendmsg()*, *setsockopt()*,
4456 *shutdown()*, *socket()*, <sys/socket.h>.
- 4457 **CHANGE HISTORY**
- 4458 First released in Issue 4.

4459 NAME

4460 setsockopt — set the socket options

4461 SYNOPSIS

```
4462 UX #include <sys/socket.h>
4463
4464 int setsockopt(int socket, int level, int option_name, const void
4465               *option_value, size_t option_len);
```

4465 DESCRIPTION

4466 The *setsockopt()* function sets the option specified by the *option_name* argument, at the protocol
 4467 level specified by the *level* argument, to the value pointed to by the *option_value* argument for the
 4468 socket associated with the file descriptor specified by the *socket* argument.

4469 The *level* argument specifies the protocol level at which the option resides. To set options at the
 4470 socket level, specify the *level* argument as SOL_SOCKET. To set options at other levels, supply
 4471 the appropriate protocol number for the protocol controlling the option. For example, to
 4472 indicate that an option will be interpreted by the TCP (Transport Control Protocol), set *level* to
 4473 the protocol number of TCP, as defined in the <netinet/in.h> header, or as determined by using
 4474 *getprotobyname()*.

4475 The *option_name* argument specifies a single option to set. The *option_name* argument and any
 4476 specified options are passed uninterpreted to the appropriate protocol module for
 4477 interpretations. The <sys/socket.h> header defines the socket level options. The socket level
 4478 options can be enabled or disabled. The options are as follows:

4479 SO_DEBUG Turns on recording of debugging information. This option enables or
 4480 disables debugging in the underlying protocol modules. This option
 4481 takes an **int** value.

4482 SO_BROADCAST Permits sending of broadcast messages, if this is supported by the
 4483 protocol. This option takes an **int** value.

4484 SO_REUSEADDR Specifies that the rules used in validating addresses supplied to *bind()*
 4485 should allow reuse of local addresses, if this is supported by the protocol.
 4486 This option takes an **int** value.

4487 SO_KEEPALIVE Keeps connections active by enabling the periodic transmission of
 4488 messages, if this is supported by the protocol. This option takes an **int**
 4489 value.

4490 If the connected socket fails to respond to these messages, the connection
 4491 is broken and processes writing to that socket are notified with a SIGPIPE
 4492 signal.

4493 SO_LINGER Lingers on a *close()* if data is present. This option controls the action
 4494 taken when unsent messages queue on a socket and *close()* is performed.
 4495 If SO_LINGER is set, the system blocks the process during *close()* until it
 4496 can transmit the data or until the time expires. If SO_LINGER is not
 4497 specified, and *close()* is issued, the system handles the call in a way that
 4498 allows the process to continue as quickly as possible. This option takes a
 4499 **linger** structure, as defined in the <sys/socket.h> header, to specify the
 4500 state of the option and linger interval.

4501 SO_OOINLINE Leaves received out-of-band data (data marked urgent) in line. This
 4502 option takes an **int** value.

4503 SO_SNDBUF Sets send buffer size. This option takes an **int** value.

- 4504 SO_RCVBUF Sets receive buffer size. This option takes an **int** value.
- 4505 For boolean options, 0 indicates that the option is disabled and 1 indicates that the option is
4506 enabled.
- 4507 Options at other protocol levels vary in format and name.
- 4508 **RETURN VALUE**
- 4509 Upon successful completion, *setsockopt()* returns 0. Otherwise, -1 is returned and *errno* is set to
4510 indicate the error.
- 4511 **ERRORS**
- 4512 The *setsockopt()* function will fail if:
- 4513 [EBADF] The *socket* argument is not a valid file descriptor.
- 4514 [EINVAL] The specified option is invalid at the specified socket level or the socket
4515 has been shut down.
- 4516 [ENOPROTOOPT] The option is not supported by the protocol.
- 4517 [ENOTSOCK] The *socket* argument does not refer to a socket.
- 4518 The *setsockopt()* function may fail if:
- 4519 [ENOMEM] There was insufficient memory available for the operation to complete.
- 4520 [ENOBUFS] Insufficient resources are available in the system to complete the call.
- 4521 [ENOSR] There were insufficient STREAMS resources available for the operation to
4522 complete.
- 4523 **APPLICATION USAGE**
- 4524 The *setsockopt()* function provides an application program with the means to control socket
4525 behaviour. An application program can use *setsockopt()* to allocate buffer space, control
4526 timeouts, or permit socket data broadcasts. The `<sys/socket.h>` header defines the socket-level
4527 options available to *setsockopt()*.
- 4528 Options may exist at multiple protocol levels. The SO_ options are always present at the
4529 uppermost socket level.
- 4530 **SEE ALSO**
- 4531 *bind()*, *endprotoent()*, *getsockopt()*, *socket()*, `<sys/socket.h>`.
- 4532 **CHANGE HISTORY**
- 4533 First released in Issue 4.

4534 **NAME**

4535 shutdown — shut down socket send and receive operations

4536 **SYNOPSIS**4537 UX `#include <sys/socket.h>`4538 `int shutdown(int socket, int how);`4539 **DESCRIPTION**4540 *socket* Specifies the file descriptor of the socket.4541 *how* Specifies the type of shutdown. The values are as follows:

4542 SHUT_RD Disables further receive operations.

4543 SHUT_WR Disables further send operations.

4544 SHUT_RDWR Disables further send and receive operations.

4545 The *shutdown()* function disables subsequent send and/or receive operations on a socket,
4546 depending on the value of the *how* argument.4547 **RETURN VALUE**4548 Upon successful completion, *shutdown()* returns 0. Otherwise, -1 is returned and *errno* is set to
4549 indicate the error.4550 **ERRORS**4551 The *shutdown()* function will fail if:4552 [EBADF] The *socket* argument is not a valid file descriptor.

4553 [ENOTCONN] The socket is not connected.

4554 [ENOTSOCK] The *socket* argument does not refer to a socket.4555 [EINVAL] The *how* argument is invalid.4556 The *shutdown()* function may fail if:4557 [ENOBUFS] Insufficient resources were available in the system to perform the
4558 operation.4559 [ENOSR] There were insufficient STREAMS resources available for the operation to
4560 complete.4561 **SEE ALSO**4562 *getsockopt()*, *read()*, *recv()*, *recvfrom()*, *recvmsg()*, *select()*, *send()*, *sendto()*, *setsockopt()*, *socket()*,
4563 *write()*, <sys/socket.h>.4564 **CHANGE HISTORY**

4565 First released in Issue 4.

4566 **NAME**

4567 socket — create an endpoint for communication

4568 **SYNOPSIS**4569 UX `#include <sys/socket.h>`4570 `int socket(int domain, int type, int protocol);`4571 **DESCRIPTION**4572 The *socket()* function creates an unbound socket in a communications domain, and returns a file
4573 descriptor that can be used in later function calls that operate on sockets.

4574 The function takes the following arguments:

4575 *domain* Specifies the communications domain in which a socket is to be created.4576 *type* Specifies the type of socket to be created.4577 *protocol* Specifies a particular protocol to be used with the socket. Specifying a
4578 *protocol* of 0 causes *socket()* to use an unspecified default protocol
4579 appropriate for the requested socket type.4580 The *domain* argument specifies the address family used in the communications domain. The
4581 address families supported by the system are implementation-dependent.4582 The `<sys/socket.h>` header defines at least the following values for the *domain* argument:

4583 AF_UNIX File system pathnames.

4584 AF_INET Internet address.

4585 The *type* argument specifies the socket type, which determines the semantics of communication
4586 over the socket. The socket types supported by the system are implementation-dependent.
4587 Possible socket types include:4588 SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-oriented byte
4589 streams, and may provide a transmission mechanism for out-of-band
4590 data.4591 SOCK_DGRAM Provides datagrams, which are connectionless, unreliable messages of
4592 fixed maximum length.4593 SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-oriented
4594 transmission path for records. A record can be sent using one or more
4595 output operations and received using one or more input operations, but a
4596 single operation never transfers part of more than one record. Record
4597 boundaries are visible to the receiver via the MSG_EOR flag.4598 If the *protocol* argument is non-zero, it must specify a protocol that is supported by the address
4599 family. The protocols supported by the system are implementation-dependent.4600 **RETURN VALUE**4601 Upon successful completion, *socket()* returns a nonnegative integer, the socket file descriptor.
4602 Otherwise a value of -1 is returned and *errno* is set to indicate the error.4603 **ERRORS**4604 The *socket()* function will fail if:

4605 [EACCES] The process does not have appropriate privileges.

4606 [EAFNOSUPPORT] The implementation does not support the specified address family.

- 4607 [EMFILE] No more file descriptors are available for this process.
- 4608 [ENFILE] No more file descriptors are available for the system.
- 4609 [EPROTONOSUPPORT]
4610 The protocol is not supported by the address family, or the protocol is not
4611 supported by the implementation.
- 4612 [EPROTOTYPE] The socket type is not supported by the protocol.
- 4613 The *socket()* function may fail if:
- 4614 [ENOBUFS] Insufficient resources were available in the system to perform the
4615 operation.
- 4616 [ENOMEM] Insufficient memory was available to fulfill the request.
- 4617 [ENOSR] There were insufficient STREAMS resources available for the operation to
4618 complete.
- 4619 **APPLICATION USAGE**
- 4620 The documentation for specific address families specify which protocols each address family
4621 supports. The documentation for specific protocols specify which socket types each protocol
4622 supports.
- 4623 The application can determine if an address family is supported by trying to create a socket with
4624 *domain* set to the protocol in question.
- 4625 **SEE ALSO**
- 4626 *accept()*, *bind()*, *connect()*, *getsockname()*, *getsockopt()*, *listen()*, *recv()*, *recvfrom()*, *recvmsg()*,
4627 *send()*, *sendmsg()*, *setsockopt()*, *shutdown()*, *socketpair()*, <**netinet/in.h**>, <**sys/socket.h**>.
- 4628 **CHANGE HISTORY**
- 4629 First released in Issue 4.

4630 **NAME**

4631 socketpair — create a pair of connected sockets

4632 **SYNOPSIS**4633 UX

```
#include <sys/socket.h>
```

4634

```
int socketpair(int domain, int type, int protocol,  
4635 int socket_vector[2]);
```

4636 **DESCRIPTION**4637 The *socketpair()* function creates an unbound pair of connected sockets in a specified *domain*, of a
4638 specified *type*, under the protocol optionally specified by the *protocol* argument. The two sockets
4639 are identical. The file descriptors used in referencing the created sockets are returned in
4640 *socket_vector*[0] and *socket_vector*[1].4641 *domain* Specifies the communications domain in which the sockets are to be
4642 created.4643 *type* Specifies the type of sockets to be created.4644 *protocol* Specifies a particular protocol to be used with the sockets. Specifying a
4645 *protocol* of 0 causes *socketpair()* to use an unspecified default protocol
4646 appropriate for the requested socket type.4647 *socket_vector* Specifies a 2-integer array to hold the file descriptors of the created socket
4648 pair.4649 The *type* argument specifies the socket type, which determines the semantics of communications
4650 over the socket. The socket types supported by the system are implementation-dependent.
4651 Possible socket types include:4652 SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-oriented byte
4653 streams, and may provide a transmission mechanism for out-of-band
4654 data.4655 SOCK_DGRAM Provides datagrams, which are connectionless, unreliable messages of
4656 fixed maximum length.4657 SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-oriented
4658 transmission path for records. A record can be sent using one or more
4659 output operations and received using one or more input operations, but a
4660 single operation never transfers part of more than one record. Record
4661 boundaries are visible to the receiver via the MSG_EOR flag.4662 If the *protocol* argument is non-zero, it must specify a protocol that is supported by the address
4663 family. The protocols supported by the system are implementation-dependent.4664 **RETURN VALUE**4665 Upon successful completion, this function returns 0. Otherwise, -1 is returned and *errno* is set to
4666 indicate the error.4667 **ERRORS**4668 The *socketpair()* function will fail if:

4669 [EAFNOSUPPORT] The implementation does not support the specified address family.

4670 [EMFILE] No more file descriptors are available for this process.

4671 [ENFILE] No more file descriptors are available for the system.

- 4672 [EOPNOTSUPP] The specified protocol does not permit creation of socket pairs.
- 4673 [EPROTONOSUPPORT]
4674 The protocol is not supported by the address family, or the protocol is not
4675 supported by the implementation.
- 4676 [EPROTOTYPE] The socket type is not supported by the protocol.
- 4677 The *socketpair()* function may fail if:
- 4678 [EACCES] The process does not have appropriate privileges.
- 4679 [ENOMEM] Insufficient memory was available to fulfill the request.
- 4680 [ENOBUFS] Insufficient resources were available in the system to perform the
4681 operation.
- 4682 [ENOSR] There were insufficient STREAMS resources available for the operation to
4683 complete.
- 4684 **APPLICATION USAGE**
- 4685 The documentation for specific address families specifies which protocols each address family
4686 supports. The documentation for specific protocols specifies which socket types each protocol
4687 supports.
- 4688 The *socketpair()* function is used primarily with UNIX domain sockets and need not be
4689 supported for other domains.
- 4690 **SEE ALSO**
- 4691 *socket()*, <sys/socket.h>.
- 4692 **CHANGE HISTORY**
- 4693 First released in Issue 4.

4694 **NAME**

4695 write, writev — write on a file

4696 **Note:** The **XSH** specification contains the basic definition of this interface. The following
4697 additional information pertains to Sockets.4698 **DESCRIPTION**4699 UX If *fdes* refers to a socket, *write()* is equivalent to *send()* with no flags set.4700 **CHANGE HISTORY**

4701 First released in Issue 4.

Sockets Headers

4702

4703
4704

This chapter describes the contents of headers used by the X/Open Sockets functions, macros and external variables.

4705
4706
4707
4708
4709

Headers contain the definition of symbolic constants, common structures, preprocessor macros and defined types. Each function in Chapter 8 specifies the headers that an application must include in order to use that function. In most cases only one header is required. These headers are present on an application development system; they do not have to be present on the target execution system.

4710 **NAME**

4711 fcntl.h — file control options

4712 **Note:** The **XSH** specification contains the basic definition of this interface. The following
4713 additional information pertains to Sockets.

4714 **DESCRIPTION**4715 UX The **<fcntl.h>** header defines the following additional values for *cmd* used by *fcntl()*:4716 **F_GETOWN** Get process or process group ID to receive SIGURG signals.4717 **F_SETOWN** Set process or process group ID to receive SIGURG signals.4718 **CHANGE HISTORY**

4719 First released in Issue 4.

4720 **NAME**

4721 sys/socket.h — Internet Protocol family

4722 **SYNOPSIS**

4723 UX #include <sys/socket.h>

4724 **DESCRIPTION**4725 The <sys/socket.h> header defines the unsigned integral type **sa_family_t** through **typedef**.4726 The <sys/socket.h> header defines the **sockaddr** structure that includes at least the following
4727 members:

4728	sa_family_t	sa_family	address family
4729	char	sa_data[]	socket address (variable-length data)

4730 The <sys/socket.h> header defines the **msghdr** structure that includes at least the following
4731 members:

4732	void	*msg_name	optional address
4733	size_t	msg_namelen	size of address
4734	struct iovec	*msg_iov	scatter/gather array
4735	int	msg_iovlen	members in msg_iov
4736	void	*msg_control	ancillary data, see below
4737	size_t	msg_controllen	ancillary data buffer len
4738	int	msg_flags	flags on received message

4739 The <sys/socket.h> header defines the **cmsghdr** structure that includes at least the following
4740 members:

4741	size_t	cmsg_len	data byte count, including hdr
4742	int	cmsg_level	originating protocol
4743	int	cmsg_type	protocol-specific type

4744 Ancillary data consists of a sequence of pairs, each consisting of a **cmsghdr** structure followed
4745 by a data array. The data array contains the ancillary data message, and the **cmsghdr** structure
4746 contains descriptive information that allows an application to correctly parse the data.4747 The values for **cmsg_level** will be legal values for the level argument to the *getsockopt()* and
4748 *setsockopt()* functions. The system documentation should specify the **cmsg_type** definitions for
4749 the supported protocols.4750 Ancillary data is also possible at the socket level. The <sys/socket.h> header defines the
4751 following macro for use as the **cmsg_type** value when **cmsg_level** is SOL_SOCKET:4752 **SCM_RIGHTS** Indicates that the data array contains the access rights to be sent or
4753 received.4754 The <sys/socket.h> header defines the following macros to gain access to the data arrays in the
4755 ancillary data associated with a message header:4756 **MSG_DATA(*cmsg*)** If the argument is a pointer to a **cmsghdr** structure, this macro returns an
4757 unsigned character pointer to the data array associated with the **cmsghdr**
4758 structure.4759 **MSG_NXTHDR(*mhdr, cmsg*)**4760 If the first argument is a pointer to a **msghdr** structure and the second
4761 argument is a pointer to a **cmsghdr** structure in the ancillary data,
4762 pointed to by the **msg_control** field of that **msghdr** structure, this macro
4763 returns a pointer to the next **cmsghdr** structure, or a null pointer if this
4764 structure is the last **cmsghdr** in the ancillary data.

4765 CMSG_FIRSTHDR(*mhdr*)
4766 If the argument is a pointer to a **msg_hdr** structure, this macro returns a
4767 pointer to the first **cmsg_hdr** structure in the ancillary data associated with
4768 this **msg_hdr** structure, or a null pointer if there is no ancillary data
4769 associated with the **msg_hdr** structure.

4770 The **<sys/socket.h>** header defines the **linger** structure that includes at least the following
4771 members:

4772 int l_onoff indicates whether linger option is enabled
4773 int l_linger linger time, in seconds

4774 The **<sys/socket.h>** header defines the following macros, with distinct integral values:

4775 SOCK_DGRAM Datagram socket
4776 SOCK_STREAM Byte-stream socket
4777 SOCK_SEQPACKET Sequenced-packet socket

4778 The **<sys/socket.h>** header defines the following macro for use as the *level* argument of
4779 *setsockopt()* and *getsockopt()*.

4780 SOL_SOCKET Options to be accessed at socket level, not protocol level.

4781 The **<sys/socket.h>** header defines the following macros, with distinct integral values, for use as
4782 the *option_name* argument in *getsockopt()* or *setsockopt()* calls:

4783 SO_DEBUG Debugging information is being recorded.
4784 SO_ACCEPTCONN Socket is accepting connections.
4785 SO_BROADCAST Transmission of broadcast messages is supported.
4786 SO_REUSEADDR Reuse of local addresses is supported.
4787 SO_KEEPALIVE Connections are kept alive with periodic messages.
4788 SO_LINGER Socket lingers on close.
4789 SO_OOBINLINE Out-of-band data is transmitted in line.
4790 SO_SNDBUF Send buffer size.
4791 SO_RCVBUF Receive buffer size.
4792 SO_ERROR Socket error status.
4793 SO_TYPE Socket type.

4794 The **<sys/socket.h>** header defines the following macros, with distinct integral values, for use as
4795 the valid values for the **msg_flags** field in the **msg_hdr** structure, or the flags parameter in
4796 *recvfrom()*, *recvmsg()*, *sendto()* or *sendmsg()* calls:

4797 MSG_TRUNC Control data truncated.
4798 MSG_EOR Terminates a record (if supported by the protocol).
4799 MSG_OOB Out-of-band data.
4800 MSG_PEEK Leave received data in queue.
4801 MSG_TRUNC Normal data truncated.
4802 MSG_WAITALL Wait for complete message.

4803 The **<sys/socket.h>** header defines the following macros, with distinct integral values:

4804 AF_UNIX UNIX domain sockets
4805 AF_INET Internet domain sockets

4806 The **<sys/socket.h>** header defines the following macros, with distinct integral values:

4807 SHUT_RD Disables further receive operations.
4808 SHUT_WR Disables further send operations.
4809 SHUT_RDWR Disables further send and receive operations.

4810 The following are declared as functions, and may also be defined as macros:

```
4811 int accept(int socket, struct sockaddr *address,  
4812           size_t *address_len);  
4813 int bind(int socket, const struct sockaddr *address,  
4814         size_t address_len);  
4815 int connect(int socket, const struct sockaddr *address,  
4816           size_t address_len);  
4817 int getpeername(int socket, struct sockaddr *address,  
4818               size_t *address_len);  
4819 int getsockname(int socket, struct sockaddr *address,  
4820               size_t *address_len);  
4821 int getsockopt(int socket, int level, int option_name,  
4822               void *option_value, size_t *option_len);  
4823 int listen(int socket, int backlog);  
4824 ssize_t recv(int socket, void *buffer, size_t length, int flags);  
4825 ssize_t recvfrom(int socket, void *buffer, size_t length,  
4826                 int flags, struct sockaddr *address, size_t *address_len);  
4827 ssize_t recvmsg(int socket, struct msghdr *message, int flags);  
4828 ssize_t send(int socket, const void *message, size_t length, int flags);  
4829 ssize_t sendmsg(int socket, const struct msghdr *message, int flags);  
4830 ssize_t sendto(int socket, const void *message, size_t length, int flags,  
4831               const struct sockaddr *dest_addr, size_t dest_len);  
4832 int setsockopt(int socket, int level, int option_name,  
4833               const void *option_value, size_t option_len);  
4834 int shutdown(int socket, int how);  
4835 int socket(int domain, int type, int protocol);  
4836 int socketpair(int domain, int type, int protocol,  
4837               int socket_vector[2]);
```

4838 **SEE ALSO**

4839 *accept()*, *bind()*, *connect()*, *getpeername()*, *getsockname()*, *getsockopt()*, *listen()*, *recv()*, *recvfrom()*,
4840 *recvmsg()*, *send()*, *sendmsg()*, *sendto()*, *setsockopt()*, *shutdown()*, *socket()*, *socketpair()*.

4841 **CHANGE HISTORY**

4842 First released in Issue 4.

4843 **NAME**

4844 sys/stat.h — data returned by the *stat()* function

4845 **Note:** The **XSH** specification contains the basic definition of this interface. The following
4846 additional information pertains to Sockets.

4847 **DESCRIPTION**

4848 UX The following additional symbolic name for the value of **st_mode** is defined:

4849 File type:

4850 S_IFMT type of file

4851 S_IFSOCK socket

4852 The following macro will test whether a file is of the specified type. The value *m* supplied to the
4853 macro is the value of **st_mode** from a **stat** structure. The macro evaluates to a non-zero value if
4854 the test is true, 0 if the test is false.

4855 S_ISSOCK (*m*) test for a socket

4856 **CHANGE HISTORY**

4857 First released in Issue 4.

4858 **NAME**4859 `sys/un.h` — definitions for UNIX-domain sockets4860 **SYNOPSIS**4861 `UX` `#include <sys/un.h>`4862 **DESCRIPTION**4863 The `<sys/un.h>` header defines the `sockaddr_un` structure that includes at least the following
4864 members:4865 `sa_family_t` `sun_family` address family
4866 `char` `sun_path[]` socket pathname4867 The `sockaddr_un` structure is used to store addresses for UNIX domain sockets. Values of this
4868 type must be cast to `struct sockaddr` for use with the socket interfaces defined in this document.4869 The `<sys/un.h>` header defines the type `sa_family_t` as described in `<sys/socket.h>`.4870 **SEE ALSO**4871 `bind()`, `socket()`, `socketpair()`.

IP Address Resolution Interfaces

4872

4873

4874

Address Resolution refers to a set of interfaces that obtain network information and are usable in conjunction with both XTI and Sockets when using the Internet Protocol (IP).

4875

4876

4877

This chapter provides reference manual pages for the address resolution API. This includes functions, macros and external variables to support application portability at the C-language source level.

4878 NAME

4879 endhostent, gethostbyaddr, gethostbyname, gethostent, sethostent — network host database
4880 functions

4881 SYNOPSIS

```
4882 UX #include <netdb.h>
4883
4884 extern int h_errno;
4885
4886 void endhostent(void);
4887
4888 struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
4889
4890 struct hostent *gethostbyname(const char *name);
4891
4892 struct hostent *gethostent(void);
4893
4894 void sethostent(int stayopen);
```

4889 DESCRIPTION

4890 The *gethostent()*, *gethostbyaddr()*, and *gethostbyname()* functions each return a pointer to a
4891 **hostent** structure, the members of which contain the fields of an entry in the network host
4892 database.

4893 The *gethostent()* function reads the next entry of the database, opening a connection to the
4894 database if necessary.

4895 The *gethostbyaddr()* function searches the database from the beginning and finds the first entry
4896 for which the address family specified by *type* matches the **h_addrtype** member and the address
4897 pointed to by *addr* occurs in *h_addrlist*, opening a connection to the database if necessary. The
4898 *addr* argument is a pointer to the binary-format (that is, not null-terminated) address in network
4899 byte order, whose length is specified by the *len* argument. The datatype of the address depends
4900 on the address family. For an address of type AF_INET, this is an **in_addr** structure, defined in
4901 **<netinet/in.h>**.

4902 The *gethostbyname()* function searches the database from the beginning and finds the first entry
4903 for which the host name specified by *name* matches the **h_name** member, opening a connection
4904 to the database if necessary.

4905 The *sethostent()* function opens a connection to the network host database, and sets the position
4906 of the next entry to the first entry. If the *stayopen* argument is non-zero, the connection to the
4907 host database will not be closed after each call to *gethostent()* (either directly, or indirectly
4908 through one of the other *gethost*()* functions).

4909 The *endhostent()* function closes the connection to the database.

4910 RETURN VALUE

4911 On successful completion, *gethostbyaddr()*, *gethostbyname()* and *gethostent()* return a pointer to a
4912 **hostent** structure if the requested entry was found, and a null pointer if the end of the database
4913 was reached or the requested entry was not found. Otherwise, a null pointer is returned.

4914 On unsuccessful completion, *gethostbyaddr()* and *gethostbyname()* functions set *h_errno* to
4915 indicate the error.

4916 ERRORS

4917 No errors are defined for *endhostent()*, *gethostent()* and *sethostent()*.

4918 The *gethostbyaddr()* and *gethostbyname()* functions will fail in the following cases, setting *h_errno*
4919 to the value shown in the list below. Any changes to *errno* are unspecified.

- 4920 [HOST_NOT_FOUND]
4921 No such host is known.
- 4922 [TRY_AGAIN] A temporary and possibly transient error occurred, such as a failure of a
4923 server to respond.
- 4924 [NO_RECOVERY] An unexpected server failure occurred which can not be recovered.
- 4925 [NO_DATA] The server recognised the request and the name but no address is
4926 available. Another type of request to the name server for the domain
4927 might return an answer.
- 4928 **APPLICATION USAGE**
- 4929 The *gethostent()*, *gethostbyaddr()*, and *gethostbyname()* functions may return pointers to static
4930 data, which may be overwritten by subsequent calls to any of these functions.
- 4931 These functions are generally used with the Internet address family.
- 4932 **SEE ALSO**
- 4933 *endservent()*, *htonl()*, *inet_addr()*, <**netdb.h**>.
- 4934 **CHANGE HISTORY**
- 4935 First released in Issue 4.

4936 **NAME**

4937 endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

4938 **SYNOPSIS**4939 UX `#include <netdb.h>`4940 `void endnetent(void);`4941 `struct netent *getnetbyaddr(in_addr_t net, int type);`4942 `struct netent *getnetbyname(const char *name);`4943 `struct netent *getnetent(void);`4944 `void setnetent(int stayopen);`4945 **DESCRIPTION**4946 The *getnetbyaddr()*, *getnetbyname()* and *getnetent()*, functions each return a pointer to a **netent**
4947 structure, the members of which contain the fields of an entry in the network database.4948 The *getnetent()* function reads the next entry of the database, opening a connection to the
4949 database if necessary.4950 The *getnetbyaddr()* function searches the database from the beginning, and finds the first entry
4951 for which the address family specified by *type* matches the **n_addrtype** member and the network
4952 number *net* matches the **n_net** member, opening a connection to the database if necessary. The
4953 *net* argument is the network number in host byte order.4954 The *getnetbyname()* function searches the database from the beginning and finds the first entry
4955 for which the network name specified by *name* matches the **n_name** member, opening a
4956 connection to the database if necessary.4957 The *setnetent()* function opens and rewinds the database. If the *stayopen* argument is non-zero,
4958 the connection to the net database will not be closed after each call to *getnetent()* (either directly,
4959 or indirectly through one of the other *getnet*()* functions).4960 The *endnetent()* function closes the database.4961 **RETURN VALUE**4962 On successful completion, *getnetbyaddr()*, *getnetbyname()* and *getnetent()*, return a pointer to a
4963 **netent** structure if the requested entry was found, and a null pointer if the end of the database
4964 was reached or the requested entry was not found. Otherwise, a null pointer is returned.4965 **ERRORS**

4966 No errors are defined.

4967 **APPLICATION USAGE**4968 The *getnetbyaddr()*, *getnetbyname()* and *getnetent()*, functions may return pointers to static data,
4969 which may be overwritten by subsequent calls to any of these functions.

4970 These functions are generally used with the Internet address family.

4971 **SEE ALSO**4972 `<netdb.h>`.4973 **CHANGE HISTORY**

4974 First released in Issue 4.

4975 **NAME**

4976 endprotoent, getprotobynumber, getprotobyname, getprotoent, setprotoent — network protocol
4977 database functions

4978 **SYNOPSIS**

4979 UX `#include <netdb.h>`

4980 `void endprotoent(void);`

4981 `struct protoent *getprotobyname(const char *name);`

4982 `struct protoent *getprotobynumber(int proto);`

4983 `struct protoent *getprotoent(void);`

4984 `void setprotoent(int stayopen);`

4985 **DESCRIPTION**

4986 The *getprotobyname()*, *getprotobynumber()* and *getprotoent()*, functions each return a pointer to a
4987 **protoent** structure, the members of which contain the fields of an entry in the network protocol
4988 database.

4989 The *getprotoent()* function reads the next entry of the database, opening a connection to the
4990 database if necessary.

4991 The *getprotobyname()* function searches the database from the beginning and finds the first entry
4992 for which the protocol name specified by *name* matches the **p_name** member, opening a
4993 connection to the database if necessary.

4994 The *getprotobynumber()* function searches the database from the beginning and finds the first
4995 entry for which the protocol number specified by *number* matches the **p_proto** member, opening
4996 a connection to the database if necessary.

4997 The *setprotoent()* function opens a connection to the database, and sets the next entry to the first
4998 entry. If the *stayopen* argument is non-zero, the connection to the network protocol database will
4999 not be closed after each call to *getprotoent()* (either directly, or indirectly through one of the other
5000 *getproto*()* functions).

5001 The *endprotoent()* function closes the connection to the database.

5002 **RETURN VALUES**

5003 On successful completion, *getprotobyname()*, *getprotobynumber()* and *getprotoent()* functions
5004 return a pointer to a **protoent** structure if the requested entry was found, and a null pointer if the
5005 end of the database was reached or the requested entry was not found. Otherwise, a null pointer
5006 is returned.

5007 **ERRORS**

5008 No errors are defined.

5009 **APPLICATION USAGE**

5010 The *getprotobyname()*, *getprotobynumber()* and *getprotoent()* functions may return pointers to
5011 static data, which may be overwritten by subsequent calls to any of these functions.

5012 These functions are generally used with the Internet address family.

5013 **SEE ALSO**

5014 `<netdb.h>`.

5015 **CHANGE HISTORY**

5016 First released in Issue 4.

5017 **NAME**

5018 endservent, getservbyport, getservbyname, getservent, setservent — network services database
5019 functions

5020 **SYNOPSIS**

```
5021 ux    #include <netdb.h>
5022
5023     void endservent(void);
5024
5025     struct servent *getservbyname(const char *name, const char *proto);
5026
5027     struct servent *getservbyport(int port, const char *proto);
5028
5029     struct servent *getservent(void);
5030
5031     void setservent(int stayopen);
```

5027 **DESCRIPTION**

5028 The *getservbyname()*, *getservbyport()* and *getservent()* functions each return a pointer to a **servent**
5029 structure, the members of which contain the fields of an entry in the network services database.

5030 The *getservent()* function reads the next entry of the database, opening a connection to the
5031 database if necessary.

5032 The *getservbyname()* function searches the database from the beginning and finds the first entry
5033 for which the service name specified by *name* matches the **s_name** member and the protocol
5034 name specified by *proto* matches the **s_proto** member, opening a connection to the database if
5035 necessary. If *proto* is a null pointer, any value of the **s_proto** member will be matched.

5036 The *getservbyport()* function searches the database from the beginning and finds the first entry
5037 for which the port specified by *port* matches the **s_port** member and the protocol name specified
5038 by *proto* matches the **s_proto** member, opening a connection to the database if necessary. If *proto*
5039 is a null pointer, any value of the **s_proto** member will be matched. The *port* argument must be
5040 in network byte order.

5041 The *setservent()* function opens a connection to the database, and sets the next entry to the first
5042 entry. If the *stayopen* argument is non-zero, the net database will not be closed after each call to
5043 the *getservent()* function (either directly, or indirectly through one of the other *getserv*()*
5044 functions).

5045 The *endservent()* function closes the database.

5046 **RETURN VALUES**

5047 On successful completion, *getservbyname()*, *getservbyport()* and *getservent()* return a pointer to a
5048 **servent** structure if the requested entry was found, and a null pointer if the end of the database
5049 was reached or the requested entry was not found. Otherwise, a null pointer is returned.

5050 **ERRORS**

5051 No errors are defined.

5052 **APPLICATION USAGE**

5053 The *port* argument of *getservbyport()* need not be compatible with the port values of all address
5054 families.

5055 The *getservent()*, *getservbyname()* and *getservbyport()* functions may return pointers to static data,
5056 which may be overwritten by subsequent calls to any of these functions.

5057 These functions are generally used with the Internet address family.

5058 **SEE ALSO**

5059 *endhostent()*, *endprotoent()*, *htonl()*, *inet_addr()*, <**netdb.h**>.

5060 **CHANGE HISTORY**

5061 First released in Issue 4.

5062 **NAME**

5063 gethostbyaddr, gethostbyname, gethostent — network host database functions

5064 **SYNOPSIS**

5065 UX #include <netdb.h>

5066 struct hostent *gethostbyaddr(const void *addr, size_t len, int type);

5067 struct hostent *gethostbyname(const char *name);

5068 struct hostent *gethostent(void);

5069 **DESCRIPTION**5070 Refer to *endhostent()*.5071 **CHANGE HISTORY**

5072 First released in Issue 4.

5073 **NAME**

5074 gethostname — get name of current host

5075 **SYNOPSIS**5076 `UX` #include <unistd.h>

5077 int gethostname(char *name, size_t namelen);

5078 **DESCRIPTION**

5079 The *gethostname()* function returns the standard host name for the current machine. The *namelen*
5080 argument specifies the size of the array pointed to by the *name* argument. The returned name is
5081 null-terminated, except that if *namelen* is an insufficient length to hold the host name, then the
5082 returned name is truncated and it is unspecified whether the returned name is null-terminated.

5083 Host names are limited to 255 bytes.

5084 **RETURN VALUE**

5085 On successful completion, 0 is returned. Otherwise, -1 is returned.

5086 **ERRORS**

5087 No errors are defined.

5088 **SEE ALSO**5089 *gethostid()* (in the XSH specification), *uname()*, <unistd.h>.5090 **CHANGE HISTORY**

5091 First released in Issue 4.

5092 **NAME**

5093 getnetbyaddr, getnetbyname, getnetent — network database functions

5094 **SYNOPSIS**5095 UX `#include <netdb.h>`5096 `struct netent *getnetbyaddr(in_addr_t net, int type);`5097 `struct netent *getnetbyname(const char *name);`5098 `struct netent *getnetent(void);`5099 **DESCRIPTION**5100 Refer to *endnetent()*.5101 **CHANGE HISTORY**

5102 First released in Issue 4.

5103 **NAME**

5104 getprotobynumber, getprotobyname, getprotoent — network protocol database functions

5105 **SYNOPSIS**5106 ux `#include <netdb.h>`5107 `struct protoent *getprotobyname(const char *name);`5108 `struct protoent *getprotobynumber(int proto);`5109 `struct protoent *getprotoent(void);`5110 **DESCRIPTION**5111 Refer to *endprotoent()*.5112 **CHANGE HISTORY**

5113 First released in Issue 4.

5114 **NAME**

5115 getservbyport, getservbyname, getservent — network services database functions

5116 **SYNOPSIS**

5117 ux #include <netdb.h>

5118 struct servent *getservbyname(const char *name, const char *proto);

5119 struct servent *getservbyport(int port, const char *proto);

5120 struct servent *getservent(void);

5121 **DESCRIPTION**5122 Refer to *endservent()*.5123 **CHANGE HISTORY**

5124 First released in Issue 4.

5125 **NAME**

5126 h_errno — error return value for network database operations

5127 **SYNOPSIS**

5128 ux extern int h_errno;

5129 **DESCRIPTION**5130 Refer to *endhostent()*.5131 **CHANGE HISTORY**

5132 First released in Issue 4.

5133 **NAME**

5134 htonl, htons, ntohl, ntohs — convert values between host and network byte order

5135 **SYNOPSIS**

5136 ux #include <arpa/inet.h>

5137 in_addr_t htonl(in_addr_t *hostlong*);5138 in_port_t htons(in_port_t *hostshort*);5139 in_addr_t ntohl(in_addr_t *netlong*);5140 in_port_t ntohs(in_port_t *netshort*);5141 **DESCRIPTION**5142 These functions convert 16-bit and 32-bit quantities between network byte order and host byte
5143 order.5144 **RETURN VALUES**5145 The *htonl()* and *htons()* functions return the argument value converted from host to network
5146 byte order.5147 The *ntohl()* and *ntohs()* functions return the argument value converted from network to host
5148 byte order.5149 **ERRORS**

5150 No errors are defined.

5151 **APPLICATION USAGE**5152 These functions are most often used in conjunction with Internet addresses and ports as
5153 returned by *gethostent()* and *getservent()*.5154 On some architectures these functions are defined as macros that expand to the value of their
5155 argument.5156 **SEE ALSO**5157 *endhostent()*, *endservent()*, <arpa/inet.h>.5158 **CHANGE HISTORY**

5159 First released in Issue 4.

5160 **NAME**

5161 inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof, inet_ntoa — Internet address
5162 manipulation

5163 **SYNOPSIS**

5164 UX `#include <arpa/inet.h>`

5165 `in_addr_t inet_addr(const char *cp);`

5166 `in_addr_t inet_lnaof(struct in_addr in);`

5167 `struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);`

5168 `in_addr_t inet_netof(struct in_addr in);`

5169 `in_addr_t inet_network(const char *cp);`

5170 `char *inet_ntoa(struct in_addr in);`

5171 **DESCRIPTION**

5172 The *inet_addr()* function converts the string pointed to by *cp*, in the Internet standard dot
5173 notation, to an integer value suitable for use as an Internet address.

5174 The *inet_lnaof()* function takes an Internet host address specified by *in* and extracts the local
5175 network address part, in host byte order.

5176 The *inet_makeaddr()* function takes the Internet network number specified by *net* and the local
5177 network address specified by *lna*, both in host byte order, and constructs an Internet address
5178 from them.

5179 The *inet_netof()* function takes an Internet host address specified by *in* and extracts the network
5180 number part, in host byte order.

5181 The *inet_network()* function converts the string pointed to by *cp*, in the Internet standard dot
5182 notation, to an integer value suitable for use as an Internet network number.

5183 The *inet_ntoa()* function converts the Internet host address specified by *in* to a string in the
5184 Internet standard dot notation.

5185 All Internet addresses are returned in network order (bytes ordered from left to right).

5186 Values specified using dot notation take one of the following forms:

5187 a.b.c.d When four parts are specified, each is interpreted as a byte of data and assigned,
5188 from left to right, to the four bytes of an Internet address.

5189 a.b.c When a three-part address is specified, the last part is interpreted as a 16-bit
5190 quantity and placed in the rightmost two bytes of the network address. This
5191 makes the three-part address format convenient for specifying Class B network
5192 addresses as *128.net.host*.

5193 a.b When a two-part address is supplied, the last part is interpreted as a 24-bit
5194 quantity and placed in the rightmost three bytes of the network address. This
5195 makes the two-part address format convenient for specifying Class A network
5196 addresses as *net.host*.

5197 a When only one part is given, the value is stored directly in the network address
5198 without any byte rearrangement.

5199 All numbers supplied as parts in dot notation may be decimal, octal, or hexadecimal, as
5200 specified in the ISO C standard (that is, a leading 0x or 0X implies hexadecimal; otherwise, a
5201 leading 0 implies octal; otherwise, the number is interpreted as decimal).

5202 RETURN VALUE

5203 Upon successful completion, *inet_addr()* returns the Internet address. Otherwise, it returns
5204 (**in_addr_t**)-1.

5205 Upon successful completion, *inet_network()* returns the converted Internet network number.
5206 Otherwise, it returns (**in_addr_t**)-1.

5207 The *inet_makeaddr()* function returns the constructed Internet address.

5208 The *inet_lnaof()* function returns the local network address part.

5209 The *inet_netof()* function returns the network number.

5210 The *inet_ntoa()* function returns a pointer to the network address in Internet-standard dot
5211 notation.

5212 ERRORS

5213 No errors are defined.

5214 APPLICATION USAGE

5215 The return value of *inet_ntoa()* may point to static data that may be overwritten by subsequent
5216 calls to *inet_ntoa()*.

5217 SEE ALSO

5218 *endhostent()*, *endnetent()*, **<arpa/inet.h>**.

5219 CHANGE HISTORY

5220 First released in Issue 4.

5221 **NAME**

5222 ntohl, ntohs — convert values between host and network byte order

5223 **SYNOPSIS**

5224 UX #include <arpa/inet.h>

5225 in_addr_t ntohl(in_addr_t *netlong*);5226 in_port_t ntohs(in_port_t *netshort*);5227 **DESCRIPTION**5228 Refer to *htonl()*.5229 **CHANGE HISTORY**

5230 First released in Issue 4.

5231 **NAME**

5232 sethostent — network host database function

5233 **SYNOPSIS**

5234 UX #include <netdb.h>

5235 void sethostent(int *stayopen*);5236 **DESCRIPTION**5237 Refer to *endhostent()*.5238 **CHANGE HISTORY**

5239 First released in Issue 4.

5240 **NAME**

5241 setnetent — network database function

5242 **SYNOPSIS**

5243 UX #include <netdb.h>

5244 void setnetent(int *stayopen*);5245 **DESCRIPTION**5246 Refer to *endnetent()*.5247 **CHANGE HISTORY**

5248 First released in Issue 4.

5249 **NAME**

5250 setprotoent — network protocol database function

5251 **SYNOPSIS**

5252 ux #include <netdb.h>

5253 void setprotoent(int *stayopen*);5254 **DESCRIPTION**5255 Refer to *endprotoent()*.5256 **CHANGE HISTORY**

5257 First released in Issue 4.

5258 **NAME**

5259 setservent — network services database function

5260 **SYNOPSIS**

5261 ux #include <netdb.h>

5262 void setservent(int *stayopen*);5263 **DESCRIPTION**5264 Refer to *endservent()*.5265 **CHANGE HISTORY**

5266 First released in Issue 4.

IP Address Resolution Headers

This chapter provides reference manual pages on the headers for the Address Resolution API.

5269 **NAME**

5270 arpa/inet.h — definitions for internet operations

5271 **SYNOPSIS**

5272 UX #include <arpa/inet.h>

5273 **DESCRIPTION**5274 The <arpa/inet.h> header defines the type **in_port_t** and the type **in_addr_t** as defined in
5275 <netinet/in.h>.5276 The <arpa/inet.h> header defines the **in_addr** structure, as defined in <netinet/in.h>.

5277 The following may be declared as functions, or defined as macros, or both:

```
5278       in_addr_t       htonl(in_addr_t hostlong);  
5279       in_port_t       htons(in_port_t hostshort);  
5280       in_addr_t       ntohl(in_addr_t netlong);  
5281       in_port_t       ntohs(in_port_t netshort);
```

5282 The following are declared as functions, and may also be defined as macros:

```
5283       in_addr_t       inet_addr(const char *cp);  
5284       in_addr_t       inet_lnaof(struct in_addr in);  
5285       struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);  
5286       in_addr_t       inet_netof(struct in_addr in);  
5287       in_addr_t       inet_network(const char *cp);  
5288       char            *inet_ntoa(struct in_addr in);
```

5289 Inclusion of the <arpa/inet.h> header may also make visible all symbols from <netinet/in.h>.

5290 **SEE ALSO**

5291 htonl(), inet_addr(), <netinet/in.h>.

5292 **CHANGE HISTORY**

5293 First released in Issue 4.

5294 **NAME**

5295 netdb.h — definitions for network database operations

5296 **SYNOPSIS**

5297 UX #include <netdb.h>

5298 **DESCRIPTION**5299 The <netdb.h> header defines the type **in_port_t** and the type **in_addr_t** as defined in
5300 <netinet/in.h>.5301 The <netdb.h> header defines the **hostent** structure that includes at least the following
5302 members:

5303	char *h_name	Official name of the host.
5304	char **h_aliases	A pointer to an array of pointers to alternative host names, 5305 terminated by a null pointer.
5306	int h_addrtype	Address type.
5307	int h_length	The length, in bytes, of the address.
5308	char **h_addr_list	A pointer to an array of pointers to network addresses (in 5309 network byte order) for the host, terminated by a null pointer.

5310 The <netdb.h> header defines the **netent** structure that includes at least the following members:

5311	char *n_name	Official, fully-qualified (including the domain) name of the host.
5312	char **n_aliases	A pointer to an array of pointers to alternative network names, 5313 terminated by a null pointer.
5314	int n_addrtype	The address type of the network.
5315	in_addr_t n_net	The network number, in host byte order.

5316 The <netdb.h> header defines the **protoent** structure that includes at least the following
5317 members:

5318	char *p_name	Official name of the protocol.
5319	char **p_aliases	A pointer to an array of pointers to alternative protocol names, 5320 terminated by a null pointer.
5321	int p_proto	The protocol number.

5322 The <netdb.h> header defines the **servent** structure that includes at least the following
5323 members:

5324	char *s_name	Official name of the service.
5325	char **s_aliases	A pointer to an array of pointers to alternative service names, 5326 terminated by a null pointer.
5327	int s_port	The port number at which the service resides, in network byte order.
5328	char *s_proto	The name of the protocol to use when contacting the service.

5329 The <netdb.h> header defines the macro **IPPORT_RESERVED** with the value of the highest
5330 reserved Internet port number.5331 The <netdb.h> header provides a declaration for *h_errno*:

5332 extern int h_errno;

5333 The <netdb.h> header defines the following macros for use as error values for *gethostbyaddr()*
5334 and *gethostbyname()*:

5335	HOST_NOT_FOUND
5336	NO_DATA
5337	NO_RECOVERY
5338	TRY_AGAIN

5339 The following are declared as functions, and may also be defined as macros:

```
5340           void                endhostent(void);
5341           void                endnetent(void);
5342           void                endprotoent(void);
5343           void                endservent(void);
5344           struct hostent       *gethostbyaddr(const void *addr, size_t len, int type);
5345           struct hostent       *gethostbyname(const char *name);
5346           struct hostent       *gethostent(void);
5347           struct netent        *getnetbyaddr(in_addr_t net, int type);
5348           struct netent        *getnetbyname(const char *name);
5349           struct netent        *getnetent(void);
5350           struct protoent      *getprotobyname(const char *name);
5351           struct protoent      *getprotobynumber(int proto);
5352           struct protoent      *getprotoent(void);
5353           struct servent       *getservbyname(const char *name, const char *proto);
5354           struct servent       *getservbyport(int port, const char *proto);
5355           struct servent       *getservent(void);
5356           void                sethostent(int stayopen);
5357           void                setnetent(int stayopen);
5358           void                setprotoent(int stayopen);
5359           void                setservent(int stayopen);
```

5360 Inclusion of the **<netdb.h>** header may also make visible all symbols from **<netinet/in.h>**.

5361 **SEE ALSO**

5362 *endhostent()*, *endnetent()*, *endprotoent()*, *endservent()*.

5363 **CHANGE HISTORY**

5364 First released in Issue 4.

5365 **NAME**

5366 netinet/in.h — Internet Protocol family

5367 **SYNOPSIS**

5368 ux #include <netinet/in.h>

5369 **DESCRIPTION**5370 The <netinet/in.h> header defines the following types through **typedef**:5371 **in_port_t** An unsigned integral type of exactly 16 bits.5372 **in_addr_t** An unsigned integral type of exactly 32 bits.5373 The <netinet/in.h> header defines the **in_addr** structure that includes at least the following
5374 member:

5375 in_addr_t s_addr

5376 The <netinet/in.h> header defines the **sockaddr_in** structure that includes at least the following
5377 member:

5378 sa_family_t sin_family

5379 in_port_t sin_port

5380 struct in_addr sin_addr

5381 unsigned char sin_zero[8]

5382 The **sockaddr_in** structure is used to store addresses for the Internet protocol family. Values of
5383 this type must be cast to **struct sockaddr** for use with the socket interfaces defined in this
5384 document.5385 The <netinet/in.h> header defines the type **sa_family_t** as described in <sys/socket.h>.5386 The <netinet/in.h> header defines the following macros for use as values of the *level* argument
5387 of *getsockopt()* and *setsockopt()*:

5388 IPPROTO_IP Dummy for IP.

5389 IPPROTO_ICMP Control message protocol.

5390 IPPROTO_TCP TCP.

5391 IPPROTO_UDP User datagram protocol.

5392 The <netinet/in.h> header defines the following macros for use as destination addresses for
5393 *connect()*, *sendmsg()* and *sendto()*:

5394 INADDR_ANY Local host address.

5395 INADDR_BROADCAST Broadcast address.

5396 **SEE ALSO**5397 *getsockopt()*, *setsockopt()*. <sys/socket.h>.5398 **CHANGE HISTORY**

5399 First released in Issue 4.

5400 **NAME**

5401 unistd.h — standard symbolic constants and types

5402 **Note:** The **XSH** specification contains the basic definition of this interface. The following
5403 additional information pertains to IP Address Resolution.5404 **DESCRIPTION**

5405 The following is declared as a function and may also be defined as a macro:

5406 int gethostname(char *address, int address_len);

5407 **SEE ALSO**5408 *gethostname()*.5409 **CHANGE HISTORY**

5410 First released in Issue 4.

ISO Transport Protocol Information

5411

5412 A.1 General

5413 This appendix describes the protocol-specific information that is relevant for ISO transport
5414 providers. This appendix also describes the protocol-specific information that is relevant when
5415 ISO transport services are provided over a TCP network⁶.

5416 In general, this Appendix describes the characteristics that the ISO and ISO-over-TCP transport
5417 providers have in common, with notes indicating where they differ.

5418 Notes:

5419 1. Protocol address:

5420 In an ISO environment, the protocol address is the transport address.

5421 2. Sending data of zero octets:

5422 The transport service definition, both in connection-oriented mode and in
5423 connectionless mode, does not permit sending a TSDU of zero octets. So, in
5424 connectionless mode, if the *len* parameter is set to zero, the *t_sndudata()* call will
5425 always return unsuccessfully with *-1* and *t_errno* set to [TBADDDATA]. In
5426 connection-oriented mode, if the *nbytes* parameter is set to zero, the *t_snd()* call
5427 will return with *-1* and *t_errno* set to [TBADDDATA] if either the T_MORE flag is
5428 set, or the T_MORE flag is not set and the preceding *t_snd()* call completed a
5429 TSDU or ETSDU (that is, the call has requested sending a zero byte TSDU or
5430 ETSDU).

5431 3. An ISO-over-TCP transport provider does not provide the connectionless mode.

5432

5433 6. The mapping for ISO-over-TCP that is referred to in this Appendix is that defined by RFC-1006: *ISO Transport Service on top of the*
5434 *TCP*, Version 3, May 1987, Marshall T Rose and Dwight E Cass, Network Working Group, Northrop Research & Technology
5435 Center. See also the *X/Open Guide to IPS-OSI Coexistence and Migration*. The relevant sections are 4.6.2 (Implementation of OSI
5436 Services over IPS) and 4.6.3 (Comments).

5437 **A.2 Options**

5438 Options are formatted according to the structure **t_opthdr** as described in Chapter 6. A
 5439 transport provider compliant to this specification supports none, all or any subset of the options
 5440 defined in Section A.2.1 and Section A.2.2 on page 194. An implementation may restrict the use
 5441 of any of these options by offering them only in the privileged or read-only mode. An ISO-over-
 5442 TCP provider supports a subset of the options defined in Section A.2.1.

5443 **A.2.1 Connection-mode Service**

5444 The protocol level of all subsequent options is ISO_TP.

5445 All options are association-related (see Chapter 6). They may be negotiated in the XTI states
 5446 T_IDLE and T_INCON, and are read-only in all other states except T_UNINIT.

5447 **A.2.1.1 Options for Quality of Service and Expedited Data**

5448 These options are all defined in the ISO 8072:1986 transport service definition (see the ISO
 5449 Transport references). The definitions are not repeated here.

Option Name	Type of Option Value	Legal Option Value	Meaning
TCO_THROUGHPUT	struct thrpt	octets per second	throughput
TCO_TRANSDEL	struct transdel	time in milliseconds	transit delay
TCO_RESERRORRATE	struct rate	OPT_RATIO	residual error rate
TCO_TRANSFFAILPROB	struct rate	OPT_RATIO	transfer failure probability
TCO_ESTFAILPROB	struct rate	OPT_RATIO	connection establ. failure probability
TCO_RELFAILPROB	struct rate	OPT_RATIO	connection release failure probability
TCO_ESTDELAY	struct rate	time in milliseconds	connection establ. delay
TCO_RELDELAY	struct rate	time in milliseconds	connection release delay
TCO_CONNRRESIL	struct rate	OPT_RATIO	connection resilience
TCO_PROTECTION	unsigned long	see text	protection
TCO_PRIORITY	unsigned long	see text	priority
TCO_EXPD	unsigned long	T_YES/T_NO	expedited data

5470 **Table A-1** Options for Quality of Service and Expedited Data

5471 OPT_RATIO is defined as $OPT_RATIO = -\log_{10}(\text{ratio})$. The *ratio* is dependent on the parameter,
 5472 but is always composed of a number of failures divided by a total number of samples. This may
 5473 be, for example, the number of TSDUs transferred in error divided by the total number of TSDU
 5474 transfers (TCO_RESERRORRATE).

5475 **Absolute Requirements**

5476 For the options in Table A-1 on page 190, the transport user can indicate whether the request is
 5477 an absolute requirement or whether a degraded value is acceptable. For the QOS options based
 5478 on **struct rate** an absolute requirement is specified via the field *minacceptvalue*, if that field is
 5479 given a value different from T_UNSPEC. The value specified for TCO_PROTECTION is an
 5480 absolute requirement if the T_ABSREQ flag is set. The values specified for TCO_EXPD and
 5481 TCO_PRIORITY are never absolute requirements.

5482 **Further Remarks**

5483 A detailed description of the options for Quality of Service can be found in the ISO 8072:1986
 5484 specification. The field elements of the structures in use for the option values are self-
 5485 explanatory. Only the following details remain to be explained.

- 5486 • If these options are returned with *t_listen()*, their values are related to the incoming
 5487 connection and not to the transport endpoint where *t_listen()* was issued. To give an
 5488 example, the value of TCO_PROTECTION is the value sent by the calling transport user, and
 5489 not the value currently effective for the endpoint (that could be retrieved by *t_optmgmt()*
 5490 with the flag T_CURRENT set). The option is not returned at all if the calling user did not
 5491 specify it. An analogous procedure applies for the other options. See also Chapter 6.

- 5492 • If, in a call to *t_accept()*, the called transport user tries to negotiate an option of higher quality
 5493 than proposed, the option is rejected and the connection establishment fails (see Section 6.3.4
 5494 on page 39).

- 5495 • The values of the QOS options TCO_THROUGHPUT, TCO_TRANSDEL,
 5496 TCO_RESERRORRATE, TCO_TRANSFFAILPROB, TCO_ESTFAILPROB,
 5497 TCO_RELFAILPROB, TCO_ESTDELAY, TCO_RELDELAY and TCO_CONNRRESIL have a
 5498 structured format. A user requesting one of these options might leave a field of the structure
 5499 unspecified by setting it to T_UNSPEC. The transport provider is then free to select an
 5500 appropriate value for this field. The transport provider may return T_UNSPEC in a field of
 5501 the structure to the user to indicate that it has not yet decided on a definite value for this
 5502 field.

5503 T_UNSPEC is not a legal value for TCO_PROTECTION, TCO_PRIORITY and TCO_EXPD.

- 5504 • TCO_THROUGHPUT and TCO_TRANSDEL
 5505 If *avgthrpt* (average throughput) is not defined (both fields set to T_UNSPEC), the transport
 5506 provider considers that the average throughput has the same values as the maximum
 5507 throughput (*maxthrpt*). An analogous procedure applies to TCO_TRANSDEL.

- 5508 • The ISO specification ISO 8073:1986 does not differentiate between average and maximum
 5509 transit delay. Transport providers that support this option adopt the values of the maximum
 5510 delay as input for the CR TPDU.

5511 • TCO_PROTECTION

5512 This option defines the general level of protection. The symbolic constants in the following
 5513 list are used to specify the required level of protection:

5514 T_NOPROTECT No protection feature.

5515 T_PASSIVEPROTECT Protection against passive monitoring.

5516 T_ACTIVEPROTECT Protection against modification, replay, addition or deletion.

5517 Both flags T_PASSIVEPROTECT and T_ACTIVEPROTECT may be set simultaneously but
 5518 are exclusive with T_NOPROTECT. If the T_ACTIVEPROTECT or T_PASSIVEPROTECT
 5519 flags are set, the user may indicate that this is an absolute requirement by also setting the

- 5520 T_ABSREQ flag.
- 5521 • TCO_PRIORITY
- 5522 Five priority levels are defined by XTI:
- 5523 T_PRIDFLT Lower level.
- 5524 T_PRILOW Low level.
- 5525 T_PRIMID Medium level.
- 5526 T_PRIHIGH High level.
- 5527 T_PRITOP Higher level.
- 5528 • An ISO-over-TCP transport provider may not support Quality of Service parameter negotiation. If not, an attempt to negotiate a Quality of Service option with an ISO-over-TCP transport provider will return with the status field set to T_NOTSUPPORT.
- 5529
- 5530
- 5531 • It is recommended that transport users avoid expedited data with an ISO-over-TCP transport provider, since the RFC 1006 treatment of expedited data does not meet the data reordering requirements specified in ISO 8072:1986, and may not be supported by the provider.
- 5532
- 5533
- 5534 The number of priority levels is not defined by ISO 8072:1986. The parameter only has meaning in the context of some management entity or structure able to judge relative importance.
- 5535

5536 A.2.1.2 Management Options

5537 These options are parameters of an ISO transport protocol according to ISO 8073:1986. They are not included in the ISO transport service definition ISO 8072:1986, but are additionally offered by XTI. Transport users wishing to be truly ISO-compliant should thus not adhere to them.

5538

5539 TCO_LTPDU is the only management option supported by an ISO-over-TCP transport provider.

5540

5541 Avoid specifying both QOS parameters and management options at the same time.

5542

5543

5544

Option Name	Type of Option Value	Legal Option Value	Meaning
TCO_LTPDU	unsigned long	length in octets	maximum length of TPDU
TCO_ACKTIME	unsigned long	time in milliseconds	acknowledge time
TCO_REASTIME	unsigned long	time in seconds	reassignment time
TCO_PREFCLASS	unsigned long	see text	preferred class
TCO_ALTCLASS1	unsigned long	see text	1st alternative class
5549 TCO_ALTCLASS2	unsigned long	see text	2nd alternative class
5550 TCO_ALTCLASS3	unsigned long	see text	3rd alternative class
5551 TCO_ALTCLASS4	unsigned long	see text	4th alternative class
5552 TCO_EXTFORM	unsigned long	T_YES/T_NO/T_UNSPEC	extended format
5553 TCO_FLOWCTRL	unsigned long	T_YES/T_NO/T_UNSPEC	flowctr
5554 TCO_CHECKSUM	unsigned long	T_YES/T_NO/T_UNSPEC	checksum
5555 TCO_NETEXP	unsigned long	T_YES/T_NO/T_UNSPEC	network expedited data
5556 TCO_NETRECPTCF	unsigned long	T_YES/T_NO/T_UNSPEC	use of network receipt confirmation
5557			
5558			

5559 **Table A-2** Management Options

5560 **Absolute Requirements**

5561 A request for any of these options is considered an absolute requirement.

5562 **Further Remarks**

5563 • If these options are returned with *t_listen()* their values are related to the incoming
5564 connection and not to the transport endpoint where *t_listen()* was issued. That means that
5565 *t_optmgmt()* with the flag T_CURRENT set would usually yield a different result (see
5566 Chapter 6).

5567 • For management options that are subject to peer-to-peer negotiation the following holds: If,
5568 in a call to *t_accept()*, the called transport user tries to negotiate an option of higher quality
5569 than proposed, the option is rejected and the connection establishment fails (see Section 6.3.4
5570 on page 39).

5571 • A connection-mode transport provider may allow the transport user to select more than one
5572 alternative class. The transport user may use the options T_ALTCLASS1, T_ALTCLASS2, etc.
5573 to denote the alternatives. A transport provider only supports an implementation-dependent
5574 limit of alternatives and ignores the rest.

5575 • The value T_UNSPEC is legal for all options in Table A-2 on page 192. It may be set by the
5576 user to indicate that the transport provider is free to choose any appropriate value. If
5577 returned by the transport provider, it indicates that the transport provider has not yet
5578 decided on a specific value.

5579 • Legal values for the options T_PREFCLASS, T_ALTCLASS1, T_ALTCLASS2, T_ALTCLASS3
5580 and T_ALTCLASS4 are T_CLASS0, T_CLASS1, T_CLASS2, T_CLASS3, T_CLASS4 and
5581 T_UNSPEC.

5582 • If a connection has been established, TCO_PREFCLASS will be set to the selected value, and
5583 T_ALTCLASS1 through T_ALTCLASS4 will be set to T_UNSPEC, if these options are
5584 supported.

5585 • *Warning* on the use of TCO_LTPDU: Sensible use of this option requires that the application
5586 programmer knows about system internals. Careless setting of either a lower or a higher
5587 value than the implementation-dependent default may degrade the performance.

5588 Legal values for an ISO transport provider are T_UNSPEC and powers of 2 between 2^{**7} and
5589 2^{**13} .

5590 Legal values for an ISO-over-TCP provider are T_UNSPEC and any power of 2 between 2^{**7}
5591 and 2^{**11} , and 65531.

5592 The action taken by a transport provider is implementation-dependent if a value is specified
5593 which is not exactly as defined in ISO 8073:1986 or its addendums.

5594 • The management options are not independent of one another, and not independent of the
5595 options defined in Section A.2.1.1 on page 190. A transport user must take care not to request
5596 conflicting values. If conflicts are detected at negotiation time, the negotiation fails according
5597 to the rules for absolute requirements (see Chapter 6). Conflicts that cannot be detected at
5598 negotiation time will lead to unpredictable results in the course of communication. Usually,
5599 conflicts are detected at the time the connection is established.

5600 Some relations that must be obeyed are:

5601 • If TCO_EXP is set to T_YES and TCO_PREFCLASS is set to T_CLASS2, TCO_FLOWCTRL
5602 must also be set to T_YES.

- 5603 • If TCO_PREFCLASS is set to T_CLASS0, TCO_EXP must be set to T_NO.
- 5604 • The value in TCO_PREFCLASS must not be lower than the value in TCO_ALTCLASS1,
5605 TCO_ALTCLASS2, and so on.
- 5606 • Depending on the chosen QOS options, further value conflicts might occur.

5607 A.2.2 Connectionless-mode Service

5608 The protocol level of all subsequent options is ISO_TP (as in Section A.2.1 on page 190).

5609 All options are association-related (see Chapter 6). They may be negotiated in all XTI states but
5610 T_UNINIT.

5611 A.2.2.1 Options for Quality of Service

5612 These options are all defined in the ISO 8072/Add.1:1986 transport service definition (see the
5613 ISO Transport references). The definitions are not repeated here. None of these options are
5614 supported by an ISO-over-TCP transport provider, since it does not support connectionless
5615 mode.

Option Name	Type of Option Value	Legal Option Value	Meaning
TCL_TRANSDEL	struct rate	time in milliseconds	transit delay
TCL_RESERRORRATE	struct rate	OPT_RATIO	residual error rate
TCL_PROTECTION	unsigned long	see text	protection
TCL_PRIORITY	unsigned long	see text	priority

5623 **Table A-3** Options for Quality of Service

5624 Absolute Requirements

5625 A request for any of these options is an absolute requirement.

5626 Further Remarks

5627 A detailed description of the options for Quality of Service can be found in ISO
5628 8072/Add.1:1986. The field elements of the structures in use for the option values are self-
5629 explanatory. Only the following details remain to be explained.

- 5630 • These options are negotiated only between the local user and the local transport provider.
- 5631 • The meaning, type of option value, and the range of legal option values are identical for
5632 TCO_RESERRORRATE and TCL_RESERRORRATE, TCO_PRIORITY and TCL_PRIORITY,
5633 TCO_PROTECTION and TCL_PROTECTION (see Table A-1 on page 190, ISO 8072:1986).
- 5634 • TCL_TRANSDEL and TCO_TRANSDEL are different. TCL_TRANSDEL specifies the
5635 maximum transit delay expected during a datagram transmission. Note that the type of
5636 option value is a **struct rate** contrary to the **struct transdel** of TCO_TRANSDEL. The range
5637 of legal option values for each field of **struct rate** is the same as that of TCO_TRANSDEL.
- 5638 • If these options are returned with *t_rcvudata()* their values are related to the received
5639 datagram and not to the transport endpoint where *t_rcvudata()* was issued. On the other
5640 hand, *t_optmgmt()* with the flag T_CURRENT set returns the values that are currently
5641 effective for outgoing datagrams.
- 5642 • The function *t_rcvuderr()* returns the option value of the data unit previously sent that
5643 produced the error.

5644 A.2.2.2 Management Options

5645 This option is a parameter of an ISO transport protocol, according to ISO 8602. It is not included
 5646 in the ISO transport service definition ISO 8072/Add.1:1986, but is an additional offer by XTI.
 5647 Transport users wishing to be truly ISO-compliant should thus not adhere to it.

5648 Avoid specifying both QOS parameters and this management option at the same time.

5649

5650

5651

5652

Option Name	Type of Option Value	Legal Option Value	Meaning
TCL_CHECKSUM	unsigned long	T_YES/T_NO	checksum computation

5653

Table A-4 Management Option

5654

Absolute Requirements

5655

A request for this option is an absolute requirement.

5656

Further Remarks

5657

5658

TCL_CHECKSUM is the option allows disabling/enabling of the checksum computation. The legal values are T_YES (checksum enabled) and T_NO (checksum disabled).

5659

5660

If this option is returned with *t_rcvdata()*, its value indicates whether or not a checksum was present in the received datagram.

5661

The advisability of turning off the checksum check is controversial.

5662 **A.3 Functions**

5663 5664	<i>t_accept()</i>	The parameter <i>call->udata.len</i> must be in the range 0 to 32. The user may send up to 32 octets of data when accepting the connection.
5665 5666 5667		If <i>fd</i> is not equal to <i>resfd</i> , <i>resfd</i> should either be in state T_UNBND or be in state T_IDLE and be bound to the same address as <i>fd</i> with the <i>qlen</i> parameter set to 0.
5668 5669 5670 5671 5672 5673 5674 5675 5676 5677		A process can listen for an incoming indication on a given <i>fd</i> and then accept the connection on another endpoint <i>resfd</i> which has been bound to the same or a different protocol address with the <i>qlen</i> parameter (of the <i>t_bind()</i> function) set to 0. The protocol address bound to the new accepting endpoint (<i>resfd</i>) should in general be the same as the listening endpoint (<i>fd</i>), because at the present time, the ISO transport service definition (ISO 8072:1986) does not authorise acceptance of an incoming connection indication with a responding address different from the called address, except under certain conditions (see ISO 8072:1986 paragraph 12.2.4, Responding Address), but it also states that it may be changed in the future.
5678	<i>t_bind()</i>	The <i>addr</i> field of the <i>t_bind()</i> structure represents the local TSAP.
5679 5680	<i>t_connect()</i>	The <i>sndcall->addr</i> structure specifies the remote called TSAP. In the present version, the returned address set in <i>rcvcall->addr</i> will have the same value.
5681 5682 5683 5684		The setting of <i>sndcall->udata</i> is optional for ISO connections, but with no data, the <i>len</i> field of <i>udata</i> must be set to 0. The <i>maxlen</i> and <i>buf</i> fields of the netbuf structure, pointed to by <i>rcvcall->addr</i> and <i>rcvcall->opt</i> , must be set before the call.
5685 5686 5687 5688 5689 5690 5691	<i>t_getinfo()</i>	The information returned by <i>t_getinfo()</i> reflects the characteristics of the transport connection or, if no connection is established, the maximum characteristics a transport connection could take on using the underlying transport provider. In all possible states except T_DATAXFER, the function <i>t_getinfo()</i> returns in the parameter <i>info</i> the same information as was returned by <i>t_open()</i> . In T_DATAXFER, however, the information returned may differ from that returned by <i>t_open()</i> , depending on:
5692 5693		— the transport class negotiated during connection establishment (ISO transport provider only)
5694		— the negotiation of expedited data transfer for this connection.
5695 5696 5697 5698		In T_DATAXFER, the <i>etsdu</i> field in the t_info structure is set to -2 if no expedited data transfer was negotiated, and to 16 otherwise. The remaining fields are set according to the characteristics of the transport protocol class in use for this connection, as defined in the table below.

5699
5700

Parameters	Before Call	After Call			
		Connection Class 0	Connection Class 1-4	Connectionless	ISO-over-TCP
<i>fd</i>	x	/	/	/	/
<i>info->addr</i>	/	x	x	x	x
<i>info->options</i>	/	x (1)	x (1)	x (1)	x (1)
<i>info->tsdu</i>	/	x (2)	x (2)	0->63488	x (2)
<i>info->etsdu</i>	/	-2	16/-2 (3)	-2	16/-2
<i>info->connect</i>	/	-2	32	-2	32/-2
<i>info->discon</i>	/	-2	64	-2	64/-2
<i>info->servtype</i>	/	T_COTS	T_COTS	T_CLTS	T_COTS
<i>info->flags</i>	/	0	0	0	0

5712

1. 'x' equals -2 or an integral number greater than zero.

5713

2. 'x' equals -1 or an integral number greater than zero.

5714

3. Depending on the negotiation of expedited data transfer.

5715 *t_listen()*

5716

5717

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()*.

5718

5719

5720

5721

5722

5723

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. So if the user queues the connect indications for too long before responding to them, the transport provider initiating the connection will disconnect it.

5724 *t_open()*

5725

5726

5727

5728

5729

5730

5731

5732

5733

The function *t_open()* is called as the first step in the initialisation of a transport endpoint. This function returns various default characteristics associated with the different classes. According to ISO 8073:1986, an OSI transport provider supports one or several out of five different transport protocols, class 0 through class 4. The default characteristics returned in the parameter *info* are those of the highest-numbered protocol class the transport provider is able to support. If, for example, a transport provider supports classes 2 and 0, the characteristics returned are those of class 2. If the transport provider is limited to class 0, the characteristics returned are those of class 0.

5734

The table below gives the characteristics associated with the different classes.

5735
5736

Parameters	Before Call	After Call			
		Connection Class 0	Connection Class 1-4	Connectionless	ISO-over-TCP
<i>name</i>	x	/	/	/	/
<i>oflag</i>	x	/	/	/	/
<i>info->addr</i>	/	x	x	x	x
<i>info->options</i>	/	x (1)	x (1)	x (1)	x (1)
<i>info->tsdu</i>	/	x (2)	x (2)	0->63488	x (2)
<i>info->etsdu</i>	/	-2	16	-2	16/-2
<i>info->connect</i>	/	-2	32	-2	32/-2
<i>info->discon</i>	/	-2	64	-2	64/-2
<i>info->servtype</i>	/	T_COTS	T_COTS	T_CLTS	T_COTS
<i>info->flags</i>	/	0	0	0	0

5737
5738

5739
5740
5741
5742
5743
5744
5745
5746
5747
5748

5749
5750

1. 'x' equals -2 or an integral number greater than zero.
2. 'x' equals -1 or an integral number greater than zero.

5751 *t_rcv()*
5752
5753
5754

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.

5755 *t_rcvconnect()*
5756
5757

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()*.

5758 *t_rcvdis()*
5759

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()*.

5760 *t_rcvudata()*
5761
5762
5763

The *unitdata->addr* structure specifies the remote TSAP. If the T_MORE flag is set, an additional *t_rcvudata()* call is needed to retrieve the entire TSDU. Only normal data is returned via the *t_rcvudata()* call. This function is not supported by an ISO-over-TCP transport provider.

5764 *t_rcvuderr()*

The *uderr->addr* structure contains the remote TSAP.

5765 *t_snd()*
5766

Zero byte TSDUs are not supported. The T_EXPEDITED flag is not a legal flag unless expedited data has been negotiated for this connection.

5767 *t_snddis()*
5768

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.

5769 *t_sndudata()*
5770
5771

The *unitdata->addr* structure specifies the remote TSAP. The ISO connectionless transport service does not support the sending of expedited data. This function is not supported by an ISO-over-TCP transport provider.

Internet Protocol-specific Information

5772

5773 B.1 General

5774 This appendix describes the protocol-specific information that is relevant for TCP and UDP
5775 transport providers.

5776 Notes

- 5777 • T_MORE flag and TSDUs

5778 The notion of TSDU is not supported by a TCP transport provider, so the T_MORE flag will
5779 be ignored when TCP is used. The TCP PUSH flag cannot be used through the XTI interface
5780 because the TCP Military Standard (see Referenced Documents) states that:

5781 “Successive pushes may not be preserved because two or more units of pushed data may be
5782 joined into a single pushed unit by either the sending or receiving TCP. Pushes are not
5783 visible to the receiving Upper Level Protocol and are not intended to serve as a record
5784 boundary marker”.

- 5785 • Expedited data

5786 TCP does not have a notion of expedited data in a sense comparable to ISO expedited data.
5787 TCP defines an urgent mechanism, by which in-line data is marked for urgent delivery. UDP
5788 has no urgent mechanism. See the TCP Military Standard for more detailed information.

- 5789 • Orderly release

5790 The orderly release functions *t_sndrel()* and *t_rcvrel()* were defined to support the orderly
5791 release facility of TCP. However, its use is not recommended so that applications using TCP
5792 may be ported to use ISO Transport. The specification of TCP states that only established
5793 connections may be closed with orderly release; that is, on an endpoint in T_DATAXFER or
5794 T_INREL state.

- 5795 • Connection establishment

5796 TCP does not allow the possibility of refusing a connection indication. Each connect
5797 indication causes the TCP transport provider to establish the connection. Therefore,
5798 *t_listen()* and *t_accept()* have a semantic which is slightly different from that for ISO
5799 providers.

5800 B.2 Options

5801 Options are formatted according to the structure **t_opthdr** as described in Chapter 6. A
 5802 transport provider compliant to this specification supports none, all or any subset of the options
 5803 defined in Section B.2.1, Section B.2.2 and Section B.2.3. An implementation may restrict the use
 5804 of any of these options by offering them only in the privileged or read-only mode.

5805 B.2.1 TCP-level Options

5806 The protocol level is INET_TCP. For this level, Table B-1 shows the options that are defined.

Option Name	Type of Option Value	Legal Option Value	Meaning
TCP_KEEPAVIVE	struct t_kpalive	see text	check if connections are alive
TCP_MAXSEG	unsigned long	length in octets	get TCP maximum segment size
TCP_NODELAY	unsigned long	T_YES/T_NO	don't delay send to coalesce packets

5813 **Table B-1** TCP-level Options

5814 These options are *not* association-related. They may be negotiated in all XTI states except
 5815 T_UNBND and T_UNINIT. They are read-only in state T_UNBND. See Chapter 6 for the
 5816 difference between options that are association-related and those that are not.

5817 Absolute Requirements

5818 A request for TCP_NODELAY and a request to activate TCP_KEEPAVIVE is an absolute
 5819 requirement. TCP_MAXSEG is a read-only option.

5820 Further Remarks

5821 TCP_KEEPAVIVE If this option is set, a keep-alive timer is activated to monitor idle
 5822 connections that might no longer exist. If a connection has been idle since
 5823 the last keep-alive timeout, a keep-alive packet is sent to check if the
 5824 connection is still alive or broken.

5825 Keep-alive packets are not an explicit feature of TCP, and this practice is
 5826 not universally accepted. According to RFC 1122:

5827 “a keep-alive mechanism should only be invoked in server applications
 5828 that might otherwise hang indefinitely and consume resources
 5829 unnecessarily if a client crashes or aborts a connection during a network
 5830 failure”.

5831 The option value consists of a structure **t_kpalive** declared as:

```
5832 struct t_kpalive {
5833     long kp_onoff; /* switch option on/off */
5834     long kp_timeout; /* keep-alive timeout in minutes */
5835 }
```

5836 Legal values for the field *kp_onoff* are:

5837	T_NO	switch keep-alive timer off
5838	T_YES	activate keep-alive timer
5839	T_YES T_GARBAGE	activate keep-alive timer and send garbage octet
5840		

5841		Usually, an implementation should send a keep-alive packet with no data (T_GARBAGE not set). If T_GARBAGE is set, the keep-alive packet contains one garbage octet for compatibility with erroneous TCP implementations.
5842		
5843		
5844		
5845		An implementation is, however, not obliged to support T_GARBAGE (see RFC 1122). Since the <i>kp_onoff</i> value is an absolute requirement, the request “T_YES T_GARBAGE” may therefore be rejected.
5846		
5847		
5848		The field <i>kp_timeout</i> determines the frequency of keep-alive packets being sent, in minutes. The transport user can request the default value by setting the field to T_UNSPEC. The default is implementation-dependent, but at least 120 minutes (see RFC 1122). Legal values for this field are T_UNSPEC and all positive numbers.
5849		
5850		
5851		
5852		
5853		The timeout value is not an absolute requirement. The implementation may pose upper and lower limits to this value. Requests that fall short of the lower limit may be negotiated to the lower limit.
5854		
5855		
5856		The use of this option might be restricted to privileged users.
5857	TCP_MAXSEG	This option is read-only. It is used to retrieve the maximum TCP segment size.
5858		
5859	TCP_NODELAY	Under most circumstances, TCP sends data as soon as it is presented. When outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgement is received. For a small number of clients, such as window systems (for example, MIT X Window System) that send a stream of mouse events which receive no replies, this packetisation may cause significant delays. TCP_NODELAY is used to defeat this algorithm. Legal option values are T_YES (“don’t delay”) and T_NO (“delay”).
5860		
5861		
5862		
5863		
5864		
5865		
5866		

5867 B.2.2 UDP-level Options

5868 The protocol level is INET_UDP. The option defined for this level is shown in Table B-2.

Option Name	Type of Option Value	Legal Option Value	Meaning
UDP_CHECKSUM	unsigned long	T_YES/T_NO	checksum computation

5873 **Table B-2** UDP-level Option

5874 This option is association-related. It may be negotiated in all XTI states except T_UNBND and T_UNINIT. It is read-only in state T_UNBND. See Chapter 6 for the difference between options that are association-related and those that are not.

5877 Absolute Requirements

5878 A request for this option is an absolute requirement.

5879 **Further Remarks**

5880 UDP_CHECKSUM The option allows disabling/enabling of the UDP checksum computation.
 5881 The legal values are T_YES (checksum enabled) and T_NO (checksum
 5882 disabled).

5883 If this option is returned with *t_rcvudata()*, its value indicates whether a
 5884 checksum was present in the received datagram or not.

5885 Numerous cases of undetected errors have been reported when
 5886 applications chose to turn off checksums for efficiency. The advisability
 5887 of ever turning off the checksum check is very controversial.

5888 **B.2.3 IP-level Options**

5889 The protocol level is INET_IP. The options defined for this level are listed in Table B-3.

Option Name	Type of Option Value	Legal Option Value	Meaning
IP_BROADCAST	unsigned int	T_YES/T_NO	permit sending of broadcast messages
IP_DONTROUTE	unsigned int	T_YES/T_NO	just use interface addresses
IP_OPTIONS	array of unsigned characters	see text	IP per-packet options
IP_REUSEADDR	unsigned int	T_YES/T_NO	allow local address reuse
IP_TOS	unsigned char	see text	IP per-packet type of service
IP_TTL	unsigned char	time in seconds	IP per packet time-to-live

5901 **Table B-3** IP-level Options

5902 IP_OPTIONS and IP_TOS are both association-related options. All other options are *not*
 5903 association-related. See Chapter 6 for the difference between association-related options and
 5904 options that are not.

5905 IP_REUSEADDR may be negotiated in all XTI states except T_UNINIT. All other options may
 5906 be negotiated in all other XTI states except T_UNBND and T_UNINIT; they are read-only in the
 5907 state T_UNBND.

5908 **Absolute Requirements**

5909 A request for any of these options is an absolute requirement.

5910 **Further Remarks**

5911 IP_BROADCAST This option requests permission to send broadcast datagrams. It was
 5912 defined to make sure that broadcasts are not generated by mistake. The
 5913 use of this option is often restricted to privileged users.

5914 IP_DONTROUTE This option indicates that outgoing messages should bypass the standard
 5915 routing facilities. It is mainly used for testing and development.

5916 IP_OPTIONS This option is used to set (retrieve) the OPTIONS field of each outgoing
 5917 (incoming) IP datagram. Its value is a string of octets composed of a
 5918 number of IP options, whose format matches those defined in the IP
 5919 specification with one exception: the list of addresses for the source
 5920 routing options must include the first-hop gateway at the beginning of
 5921 the list of gateways. The first-hop gateway address will be extracted from

5922 the option list and the size adjusted accordingly before use.

5923 The option is disabled if it is specified with “no value”; that is, with an
5924 option header only.

5925 The functions `t_connect()` (in synchronous mode), `t_listen()`,
5926 `t_rcvconnect()` and `t_rcvudata()` return the OPTIONS field, if any, of the
5927 received IP datagram associated with this call. The function `t_rcvuderr()`
5928 returns the OPTIONS field of the data unit previously sent that produced
5929 the error. The function `t_optmgmt()` with T_CURRENT set retrieves the
5930 currently effective IP_OPTIONS that is sent with outgoing datagrams.

5931 Common applications never need this option. It is mainly used for
5932 network debugging and control purposes.

5933 IP_REUSEADDR Many TCP implementations do not allow the user to bind more than one
5934 transport endpoint to addresses with identical port numbers. If
5935 IP_REUSEADDR is set to T_YES this restriction is relaxed in the sense
5936 that it is now allowed to bind a transport endpoint to an address with a
5937 port number and an underspecified internet address (“wild card”
5938 address) and further endpoints to addresses with the same port number
5939 and (mutually exclusive) fully specified internet addresses.

5940 IP_TOS This option is used to set (retrieve) the *type-of-service* field of an outgoing
5941 (incoming) IP datagram. This field can be constructed by any OR'ed
5942 combination of one of the precedence flags and the type-of-service flags
5943 T_LDELAY, T_HITHRPT and T_HIREL:

5944 — Precedence:

5945 These flags specify datagram precedence, allowing senders to indicate
5946 the importance of each datagram. They are intended for Department
5947 of Defense applications. Legal flags are:

5948 T_ROUTINE
5949 T_PRIORITY
5950 T_IMMEDIATE
5951 T_FLASH
5952 T_OVERRIDEFLASH
5953 T_CRITIC_ECP
5954 T_INETCONTROL
5955 T_NETCONTROL.

5956 Applications using IP_TOS but not the precedence level should use
5957 the value T_ROUTINE for precedence.

5958 — Type of service:

5959 These flags specify the type of service the IP datagram desires. Legal
5960 flags are:

5961 T_NOTOS requests no distinguished type of service
5962 T_LDELAY requests low delay
5963 T_HITHRPT requests high throughput
5964 T_HIREL requests high reliability

5965 The option value is set using the macro `SET_TOS(prec,tos)`, where *prec* is
5966 set to one of the precedence flags and *tos* to one or an OR'ed combination
5967 of the type-of-service flags. `SET_TOS()` returns the option value.

5968 The functions *t_connect()*, *t_listen()*, *t_rcvconnect()* and *t_rcvudata()*
5969 return the *type-of-service* field of the received IP datagram associated with
5970 this call. The function *t_rcvuderr()* returns the *type-of-service* field of the
5971 data unit previously sent that produced the error.

5972 The function *t_optmgmt()* with T_CURRENT set retrieves the currently
5973 effective IP_TOS value that is sent with outgoing datagrams.

5974 The requested *type-of-service* cannot be guaranteed. It is a hint to the
5975 routing algorithm that helps it choose among various paths to a
5976 destination. Note also, that most hosts and gateways in the Internet these
5977 days ignore the *type-of-service* field.

5978 IP_TTL This option is used to set the *time-to-live* field in an outgoing IP datagram.
5979 It specifies how long, in seconds, the datagram is allowed to remain in the
5980 Internet. The *time-to-live* field of an incoming datagram is not returned by
5981 any function (since it is not an association-related option).

5982 **B.3 Functions**

5983	<i>t_accept()</i>	Issuing <i>t_accept()</i> assigns an already established connection to <i>resfd</i> .
5984		Since user data cannot be exchanged during the connection establishment phase, <i>call->udata.len</i> must be set to 0. Also, <i>resfd</i> must be bound to the same address as <i>fd</i> . A potential restriction on binding of endpoints to protocol addresses is described under <i>t_bind()</i> below.
5985		
5986		
5987		
5988		If association-related options (IP_OPTIONS, IP_TOS) are to be sent with the connect confirmation, the values of these options must be set with <i>t_optmgmt()</i> before the T_LISTEN event occurs. When the transport user detects a T_LISTEN, TCP has already established the connection. Association-related options passed with <i>t_accept()</i> become effective at once, but since the connection is already established, they are transmitted with subsequent IP datagrams sent out in the T_DATAXFER state.
5989		
5990		
5991		
5992		
5993		
5994		
5995	<i>t_bind()</i>	The <i>addr</i> field of the t_bind structure represents the local socket; that is, an address which specifically includes a port identifier.
5996		
5997		In the connection-oriented mode (that is, TCP), the <i>t_bind()</i> function may only bind one transport endpoint to any particular protocol address. If that endpoint was bound in passive mode; that is, <i>qlen</i> > 0, then other endpoints will be bound to the passive endpoint's protocol address via the <i>t_accept()</i> function only; that is, if <i>fd</i> refers to the passive endpoint and <i>resfd</i> refers to the new endpoint on which the connection is to be accepted, <i>resfd</i> will be bound to the same protocol address as <i>fd</i> after the successful completion of the <i>t_accept()</i> function.
5998		
5999		
6000		
6001		
6002		
6003		
6004		
6005	<i>t_connect()</i>	The <i>sndcall->addr</i> structure specifies the remote socket. In the present version, the returned address set in <i>rcvcall->addr</i> will have the same value. Since user data cannot be exchanged during the connection establishment phase, <i>sndcall->udata.len</i> must be set to 0.
6006		
6007		
6008		
6009		Note that the peer TCP, and not the peer transport user, confirms the connection.
6010		
6011	<i>t_listen()</i>	Upon successful return, <i>t_listen()</i> indicates an existing connection and not a connection indication.
6012		
6013		Since user data cannot be exchanged during the connection establishment phase, <i>call->udata.maxlen</i> must be set to 0 before the call to <i>t_listen()</i> . The <i>call->addr</i> structure contains the remote calling socket.
6014		
6015		
6016	<i>t_look()</i>	As soon as a segment with the TCP urgent pointer set enters the TCP receive buffer, the event T_EXDATA is indicated. T_EXDATA remains set until all data up to the byte pointed to by the TCP urgent pointer has been received. If the urgent pointer is updated, and the user has not yet received the byte previously pointed to by the urgent pointer, the update is invisible to the user.
6017		
6018		
6019		
6020		
6021	<i>t_open()</i>	<i>t_open()</i> is called as the first step in the initialisation of a transport endpoint. This function returns various default characteristics of the underlying transport protocol by setting fields in the t_info structure.
6022		
6023		
6024		The following should be the values returned by the call to <i>t_open()</i> and <i>t_getinfo()</i> with the indicated transport providers.
6025		

6026
6027
6028
6029
6030
6031
6032
6033
6034
6035
6036
6037
6038

Parameters	Before call	After call	
		TCP/IP	UDP/IP
<i>name</i>	x	/	/
<i>oflag</i>	x	/	/
<i>info->addr</i>	/	x	x
<i>info->options</i>	/	x	x
<i>info->tsdu</i>	/	0	x
<i>info->etsdu</i>	/	-1	-2
<i>info->connect</i>	/	-2	-2
<i>info->discon</i>	/	-2	-2
<i>info->servtype</i>	/	T_COTS/T_COTS_ORD	T_CLTS
<i>info->flags</i>	/	T_SNDZERO	T_SNDZERO

6039

'x' equals -2 or an integral number greater than zero.

6040 *t_rcv()*

6041
6042
6043
6044
6045
6046
6047
6048

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

6049 *t_rcvconnect()*

6050
6051

Since user data cannot be exchanged during the connection establishment phase, *call->udata.maxlen* must be set to 0 before the call to *t_rcvconnect()*. On return, the *call->addr* structure contains the remote calling socket.

6052 *t_rcvdis()*

6053

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

6054 *t_snd()*

6055
6056
6057

The T_MORE flag should be ignored. If *t_snd()* is called with more than one byte specified and with the T_EXPEDITED flag set, then the last byte of the buffer will be the byte pointed to by the TCP urgent pointer. If the T_EXPEDITED flag is set, at least one byte must be sent.

6058 *Implementor's Note: Data for a t_snd() call with the T_EXPEDITED flag set may not pass data sent previously.*

6060 *t_snddis()*

6061

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

6062 *t_sndudata()*

6063

Be aware that the maximum size of a connectionless TSDU varies among implementations.

Guidelines for Use of XTI

6064

6065 C.1 Transport Service Interface Sequence of Functions

6066 In order to describe the allowable sequence of function calls, this section gives some rules
6067 regarding the maintenance of the state of the interface:

- 6068 • It is the responsibility of the transport provider to keep a record of the state of the interface as
6069 seen by the transport user.
- 6070 • The transport provider will not process a function that places the interface out of state.
- 6071 • If the user issues a function out of sequence, the transport provider will indicate this where
6072 possible through an error return on that function. The state will not change. In this case, if
6073 any data is passed with the function when not in the T_DATAXFER state, that data will not
6074 be accepted or forwarded by the transport provider.
- 6075 • The uninitialised state (T_UNINIT) of a transport endpoint is the initial state. The endpoint
6076 must be initialised and bound before the transport provider may view it as active.
- 6077 • The uninitialised state is also the final state, and the transport endpoint must be viewed as
6078 unused by the transport provider. The *t_close()* function will close the transport endpoint
6079 and free the transport library resources for another endpoint.
- 6080 • According to Table 5-5 on page 32, *t_close()* should only be issued from the T_UNBND state.
6081 If it is issued from any other state, and no other user has that endpoint open, the action will
6082 be abortive, the transport endpoint will be successfully closed, and the library resources will
6083 be freed for another endpoint. When *t_close()* is issued, the transport provider must ensure
6084 that the address associated with the specified transport endpoint has been unbound from
6085 that endpoint. The provider sends appropriate disconnects if *t_close()* is not issued from the
6086 unbound state.

6087 The following rules apply only to the connection-mode transport service:

- 6088 • The transport connection release phase can be initiated at any time during the connection
6089 establishment phase or data transfer phase.
- 6090 • The only time the state of a transport service interface of a transport endpoint may be
6091 transferred to another transport endpoint is when the *t_accept()* function specifies such
6092 action. The following rules then apply to the cooperating transport endpoints:
 - 6093 — The endpoint that is to accept the current state of the interface must be bound to an
6094 appropriate protocol address and must be in the T_IDLE state.
 - 6095 — The user transferring the current state of an endpoint must have correct permissions for
6096 the use of the protocol address bound to the accepting transport endpoint.
 - 6097 — The endpoint that transfers the state of the transport interface is placed into the T_IDLE
6098 state by the transport provider after the completion of the transfer if there are no more
6099 outstanding connect indications.

6100 C.2 Example in Connection-oriented Mode

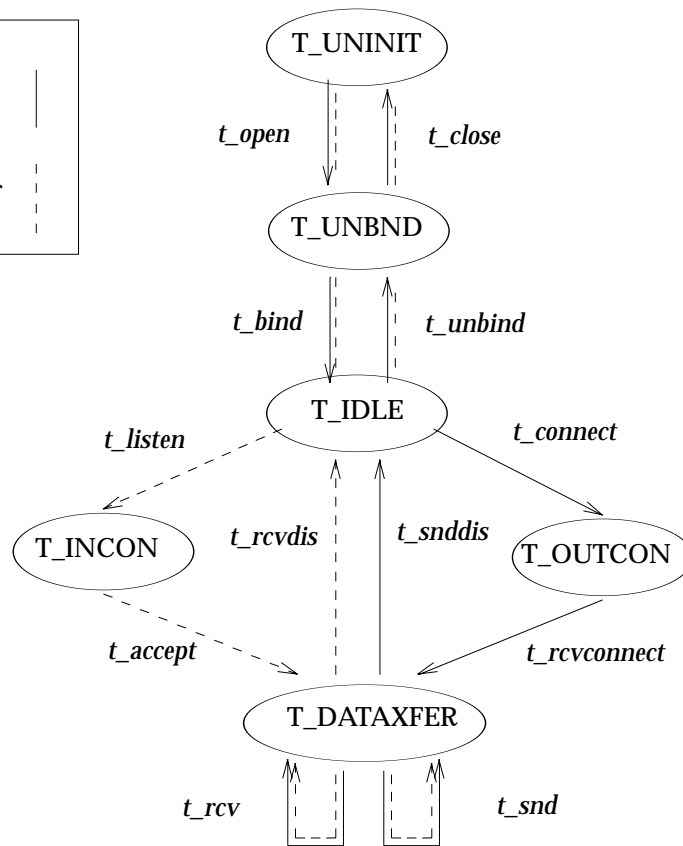
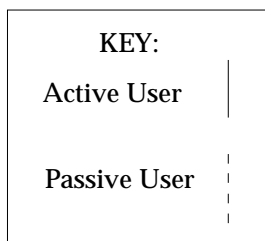
6101 Table C-1 on page 209 shows the allowable sequence of functions of an active user and passive
6102 user communicating using a connection-mode transport service. This example is not meant to
6103 show all the functions that must be called, but rather to highlight the important functions that
6104 request a particular service. Blank lines are used to indicate that the function would be called by
6105 another user prior to a related function being called by the remote user. For example, the active
6106 user calls *t_connect()* to request a connection and the passive user would receive an indication of
6107 the connect request (via the return from *t_listen()*) and then would call the *t_accept()*.

6108 The state diagram in Table C-1 on page 209 shows the flow of the events through the various
6109 states. The active user is represented by a solid line and the passive user is represented by a
6110 dashed line. This example shows a successful connection being established and terminated
6111 using connection-mode transport service without orderly release. For a detailed description of
6112 all possible states and events, see Table 5-7 on page 33.

6113
6114
6115
6116
6117
6118
6119
6120
6121
6122
6123
6124
6125
6126

Active User	Passive User
<i>t_open()</i>	<i>t_open()</i>
<i>t_bind()</i>	<i>t_bind()</i>
	<i>t_listen()</i>
<i>t_connect()</i>	
	<i>t_accept()</i>
<i>t_rcvconnect()</i>	
<i>t_snd()</i>	
	<i>t_rcv()</i>
<i>t_snddis()</i>	
	<i>t_rcvdis()</i>
<i>t_unbind()</i>	<i>t_unbind()</i>
<i>t_close()</i>	<i>t_close()</i>

6127
6128
6129



6130

6131

Table C-1 Sequence of Transport Functions in Connection-oriented Mode

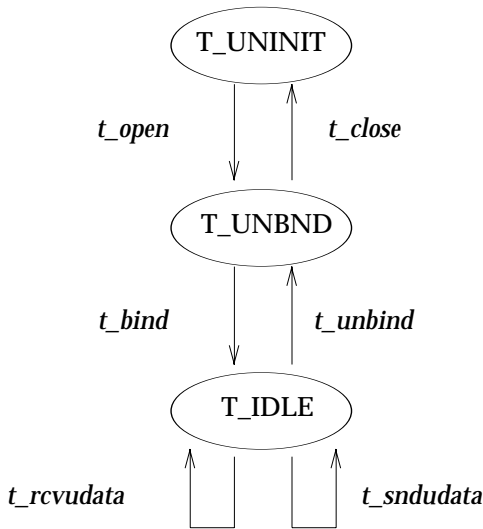
6132 **C.3 Example in Connectionless Mode**

6133 Table C-2 shows the allowable sequence of functions of user A and user B communicating using
 6134 a connectionless transport service. This example is not meant to show all the functions that
 6135 must be called but rather to highlight the important functions that request a particular service.
 6136 Blank lines are used to indicate that a function would be called by another user prior to a related
 6137 function being called by the remote user.

6138 The state diagram that follows shows the flow of the events through the various states. This
 6139 example shows a successful exchange of data between user A and user B. For a detailed
 6140 description of all possible states and events, see Table 5-7 on page 33.

6141
 6142
 6143
 6144
 6145
 6146
 6147
 6148

User A	User B
<i>t_open()</i>	<i>t_open()</i>
<i>t_bind()</i>	<i>t_bind()</i>
<i>t_sndudata()</i>	
	<i>t_rcvudata()</i>
<i>t_unbind()</i>	<i>t_unbind()</i>
<i>t_close()</i>	<i>t_close()</i>



6149

6150 **Table C-2** Sequence of Transport Functions in Connectionless Mode

6151 C.4 Writing Protocol-independent Software

6152 In order to maximise portability of XTI applications between different kinds of machine and to
6153 support protocol independence, there are some general rules:

- 6154 1. An application should only make use of those functions and mechanisms described as
6155 being mandatory features of XTI.
- 6156 2. In the connection-mode service, the concept of a transport service data unit (TSDU) may
6157 not be supported by all transport providers. The user should make no assumptions about
6158 the preservation of logical data boundaries across a connection.
- 6159 3. If an application is not intended to run only over an ISO transport provider, then the name
6160 of the device should not be hard-coded into it. While software may be written for a
6161 particular class of service (for example, connectionless-mode service), it should not be
6162 written to depend on any attribute of the underlying protocol.
- 6163 4. The protocol-specific service limits returned on the *t_open()* and *t_getinfo()* functions must
6164 not be exceeded. It is the responsibility of the user to access these limits and then adhere to
6165 the limits throughout the communication process.
- 6166 5. The user program should not look at or change options that are specific to the underlying
6167 protocol. The *t_optmgmt()* function enables a user to access default protocol options from
6168 the transport provider, which may then be blindly passed as an argument on the
6169 appropriate connect establishment function. Optionally, the user can choose not to pass
6170 options as an argument on connect establishment functions.
- 6171 6. Protocol-specific addressing issues should be hidden from the user program. Similarly, the
6172 user must have some way of accessing destination addresses in an invisible manner, such
6173 as through a name server. However, the details for doing so are outside the scope of this
6174 interface specification.
- 6175 7. The reason codes associated with *t_rcvdis()* are protocol-dependent. The user should not
6176 interpret this information if protocol independence is a concern.
- 6177 8. The error codes associated with *t_rcvuderr()* are protocol-dependent. The user should not
6178 interpret this information if protocol independence is a concern.
- 6179 9. The optional orderly release facility of the connection-mode service (that is, *t_sndrel()* and
6180 *t_rcvrel()*) should not be used by programs targeted for multiple protocol environments.
6181 This facility is not supported by all connection-based transport protocols. In particular, its
6182 use will prevent programs from successfully communicating with ISO open systems.
- 6183 10. The semantics of expedited data are different across different transport providers (for
6184 example, ISO and TCP). An application intended to run over different transport providers
6185 should avoid their use.

6186 C.5 Event Management

6187 In the absence of a standardised Event Management interface, the following guidelines are
6188 offered for the use of existing and widely available mechanisms by XTI applications.

6189 These guidelines provide information additional to that given in Section 3.7 on page 14 and
6190 Section 3.8 on page 16.

6191 For applications to use XTI in a fully asynchronous manner, they will need to use the facilities of
6192 an Event Management (EM) Interface. Such an EM will allow the application to be notified of a
6193 number of XTI events over a range of active endpoints. These events may be associated with:

- 6194 • connection indication
- 6195 • data indication
- 6196 • disconnection indication
- 6197 • flow control being lifted.

6198 In the same way, the EM mechanism should allow the application to be notified of events
6199 coming from external sources, such as:

- 6200 • asynchronous I/O completion
- 6201 • expiration of timer
- 6202 • resource availability.

6203 When handling multiple transport connections, the application could either:

- 6204 • fork a process for each new connection to be handled
- 6205 or:
- 6206 • handle all connections within a single process by making use of the EM facilities.

6207 The application will have to maintain an appropriate balance and choose the right trade-off
6208 between the number of processes and the number of connections managed per process in order
6209 to minimise the resulting overhead.

6210 Unfortunately, the system facilities to suspend and await notification of an event are presently
6211 system-dependent, although work is in progress within standards bodies to provide a unified
6212 and portable mechanism.

6213 Hence, for the foreseeable future, applications could use whatever underlying system facilities
6214 exist for event notification.

6215 C.5.1 Short-term Solution

6216 Many vendors currently provide either the System V *poll()* or BSD *select()* system calls which
6217 both give the ability to suspend until there is activity on a member of a set of file descriptors or a
6218 timeout.

6219 Given the fact that a transport endpoint identifying a transport connection maps to a file
6220 descriptor, applications can take advantage of such EM mechanisms offered by the system (for
6221 example, *poll()* or *select()*). The design of more efficient and sophisticated applications, that
6222 make full use of all the XTI features, then becomes easily possible.

6223 Guidelines for the use of *poll()* and *select()* are included in manual-page format, following the
6224 end of this section.

6225 **C.5.2 XTI Events**

6226 The XTI events can be divided into two classes of events.

- **Class 1:** events related to reception of data.

6227
6228
6229
6230
6231
6232
6233
6234
6235

T_LISTEN	Connect request indication.
T_CONNECT	Connect response indication.
T_DATA	Reception of normal data indication.
T_EXDATA	Reception of expedited data indication.
T_DISCONNECT	Disconnect request indication.
T_ORDREL	Orderly release request indication.
T_UDERR	Notification of an error in a previously sent datagram.

6236 This class of events should always be monitored by the application.

- **Class 2:** events related to emission of data (flow control).

6237
6238
6239
6240

T_GODATA	Normal data may be sent again.
T_GOEXDATA	Expedited data may be sent again.

6241 This class of events informs the application that flow control restrictions have been lifted on
6242 a given file descriptor.

6243 The application should request to be notified of this class of events whenever a flow control
6244 restriction has previously occurred on this endpoint (for example, [TFLOW] error has been
6245 returned on a *t_snd()* call).

6246 Note that this class of event should not be monitored systematically otherwise the
6247 application would be notified each time a message is sent.

6248 C.6 The Poll Function

6249 *poll()* is defined in the System V Interface Definition, Third Edition as follows. Note that this
6250 definition may vary slightly in other systems.

6251 UX If the implementation defines `_XOPEN_UNIX`, refer to the description of *poll()* in the **XSH**
6252 specification. Moreover, Chapter 8 on page 105 of the current document gives additional
6253 information on the specific effect of *poll()* when applied to Sockets.

6254 The manual page definition on the next page is followed by a section giving guidelines for use of
6255 System V *poll()*.

6256 NAME

6257 poll - input/output multiplexing

6258 SYNOPSIS

```
6259 #include <poll.h>
6260 int poll(struct pollfd fds[], unsigned long nfds, int timeout);
```

6261 DESCRIPTION

6262 *poll()* provides users with a mechanism for multiplexing input/output over a set of file
 6263 descriptors. *poll()* identifies those file descriptors on which a user can read or write data, or on
 6264 which certain events have occurred. A user can read data using *read()* and write data using
 6265 *write()*. For STREAMS file descriptors, a user can also receive messages using *getmsg()* and
 6266 *getpmsg()*, and send messages using *putmsg()* and *putpmsg()*.

6267 *fds* specifies the file descriptors to be examined and the events of interest for each file descriptor.
 6268 It is a pointer to an array with one element for each open file descriptor of interest. The array's
 6269 elements are *pollfd* structures which contain the following members:

```
6270 int fd;           /* file descriptor */
6271 short events;    /* requested events */
6272 short revents;   /* returned events */
```

6273 where *fd* specifies an open file descriptor and *events* and *revents* are bit-masks constructed by
 6274 OR'ing a combination of the following event flags:

6275	POLLIN	Data other than high-priority data may be read without blocking. For
6276		STREAMS, this flag is set even if the message is of zero length.
6277	POLLRDNORM	Normal data (priority band equals 0) may be read without blocking. For
6278		STREAMS, this flag is set even if the message is of zero length.
6279	POLLRDBAND	Data from a non-zero priority band may be read without blocking. For
6280		STREAMS, this flag is set even if the message is of zero length.
6281	POLLPRI	High-priority data may be received without blocking. For STREAMS, this flag
6282		is set even if the message is of zero length.
6283	POLLOUT	Normal data may be written without blocking.
6284	POLLWRBAND	Priority data (priority band greater than 0) may be written.
6285	POLLER	An error has occurred on the device or STREAM. This flag is only valid in the
6286		<i>revents</i> bitmask; it is not used in the <i>events</i> field.
6287	POLLUP	The device has been disconnected. This event and POLLOUT are mutually
6288		exclusive; a STREAM can never be writable if a hangup has occurred.
6289		However, this event and POLLIN, POLLRDNORM, POLLRDBAND or
6290		POLLPRI are not mutually exclusive. This flag is only valid in the <i>revents</i>
6291		bitmask; it is not used in the <i>events</i> field.
6292	POLLNVAL	The specified <i>fd</i> value is invalid. This flag is only valid in the <i>revents</i> field; it is
6293		not used in the <i>events</i> field.

6294 For each element of the array pointed to by *fds*, *poll()* examines the given file descriptor for the
 6295 event(s) specified in *events*. The number of file descriptors to be examined is specified by *nfds*.

6296 If the value of *fd* is less than zero, *events* is ignored and *revents* is set to zero in that entry on
 6297 return from *poll()*.

6298 The results of the *poll()* query are stored in the *revents* field in the *pollfd* structure. Bits are set in
 6299 the *revents* bitmask to indicate which of the requested events are true. If none of the requested

6300 events are true, none of the specified bits is set in *revents* when the *poll()* call returns. The events
6301 flags POLLUP, POLLERR and POLLNVAL, are always set in the *revents* if the conditions they
6302 indicate are true; this occurs even though these flags were not present in *events*.

6303 If none of the defined events have occurred on any selected file descriptor, *poll()* waits at least
6304 *timeout* milliseconds for an event to occur on any of the selected file descriptors. On a computer
6305 where millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal
6306 value available on that system. If the value of *timeout* is 0, *poll()* returns immediately. If the
6307 value of *timeout* is -1 *poll()* blocks until a requested event occurs or until the call is interrupted.
6308 *poll()* is not affected by the O_NDELAY and O_NONBLOCK flags.

6309 RETURN VALUES

6310 Upon successful completion, the function *poll()* returns a non-negative value. A positive value
6311 indicates the total number of file descriptors that have been selected (that is, file descriptors for
6312 which the *revents* field is non-zero). A value of 0 indicates that the call timed out and no file
6313 descriptors have been selected. Upon failure, the function *poll()* returns a value -1 and sets *errno*
6314 to indicate an error.

6315 ERRORS

6316 Under the following conditions, the function *poll()* fails and sets *errno* to:

6317 EAGAIN If the allocation of internal data structures failed but the request should be
6318 attempted again.

6319 EINTR If a signal was caught during the *poll()* system call.

6320 EINVAL If the argument *nfds* is less than zero or greater than {OPEN_MAX}.

6321 **C.7 Use of Poll**

6322 For an application to be notified of any XTI events on each of its active endpoints, the array
 6323 pointed to by *fds* should contain as many elements as active endpoints identified by the file
 6324 descriptor *fd*, and the *events* member of those elements should be set to the combination of event
 6325 flags as specified below:

6326 • For Class 1 events:

6327 POLLIN | POLLPRI (for System V Release 3)

6328 or:

6329 POLLIN | POLLRDNORM | POLLRDBAND | POLLPRI (for System V Release 4).

6330 • For Class 2 events:

6331 POLLOUT (for System V Release 3)

6332 or:

6333 POLLOUT | POLLWRBAND (for System V Release 4).

6334 In a System V Release 3, the meaning of POLLOUT may differ for different XTI
 6335 implementations. It could either mean:

6336 • that both normal and expedited data may be sent

6337 or:

6338 • that normal data may be sent and the flow of expedited data cannot be monitored via *poll()*.

6339 A truly portable XTI application should, therefore, not assume that the flow of expedited data is
 6340 monitored by *poll()*. This is not a serious restriction, since an application usually only sends
 6341 small amounts of expedited data and flow restrictions are not a major problem.

6342 In a System V Release 4, the meaning of POLLOUT and POLLWRBAND is intended to be the
 6343 same for all XTI implementations.

6344 POLLOUT Normal data may be sent.

6345 POLLWRBAND Expedited data may be sent.

6346 The following description gives the outline of an XTI server program making use of the System
 6347 V *poll()*.

```

6348  /*
6349  * This is a simple server application example to show how poll() can
6350  * be used in a portable manner to wait for the occurrence of XTI events.
6351  * In this example, poll() is used to wait for the events T_LISTEN,
6352  * T_DISCONNECT, T_DATA and T_GODATA.
6353  * The number of poll flags has increased from System V Release 3 to
6354  * System V Release 4. Hence, if this program is to be used in a
6355  * System V Release 3, the constant SVR3 must be defined during
6356  * compile time.
6357  *
6358  * A transport endpoint is opened in asynchronous mode over a
6359  * message-oriented transport provider (for example, ISO). The endpoint
6360  * is bound with qlen = 1 and the application enters an endless loop
6361  * to wait for all incoming XTI events on all its active endpoints.
6362  * For all connect indications received, a new endpoint is opened
6363  * with qlen = 0 and the connect request is accepted on that endpoint.
6364  * For all established connections, the application waits for data
6365  * to be received from one of its clients, sends the received data
6366  * back to the sender and waits for data again.
6367  * The cycle repeats until all the connections are released by
6368  * the clients. The disconnect indications are processed and the
6369  * endpoints closed.
6370  *
6371  * The example references two fictitious functions:
6372  *
6373  * - int get_provider(int tpid, char * tpname)
6374  *     Given a number as transport provider id, the function returns in
6375  *     tpname a string as transport provider name that can be used with
6376  *     t_open(). This function hides the different naming schemes of
6377  *     different XTI implementations.
6378  *
6379  * - int get_address(char * symb_name, struct netbuf address)
6380  *     Given a symbolic name symb_name and a pointer to a struct netbuf
6381  *     with allocated buffer space as input, the function returns a
6382  *     protocol address. This function hides the different addressing
6383  *     schemes of different XTI implementations.
6384  */

6385  /*
6386  * General Includes
6387  */
6388  #include <sys/types.h>
6389  #include <fcntl.h>
6390  #include <stdio.h>
6391  #include <xti.h>

6392  /*
6393  * Include files for poll()
6394  */
6395  #include <stropts.h>
6396  #include <poll.h>

6397  /*
6398  * Various Defines
6399  */
6400  /*
6401  * The XTI events T_CONNECT, T_DISCONNECT, T_LISTEN, T_ORDREL and T_UDERR
6402  * are related to one of the poll flags in INEVENTS (to which one, depends
6403  * on the implementation). POLLOUT means that (at least) normal data may

```

```

6404     * be sent, and POLLWRBAND that expedited data may be sent.
6405     */

6406     #ifdef SVR3
6407     #define ERREVENTS      (POLLERR | POLLHUP | POLLNVAL)
6408     #define INEVENTS      (POLLIN | POLLPRI)
6409     #define OUTEVENTS     POLLOUT
6410     #else
6411     #define ERREVENTS      (POLLERR | POLLHUP | POLLNVAL)
6412     #define INEVENTS      (POLLIN | POLLRDNORM | POLLRDBAND | POLLPRI)
6413     #define OUTEVENTS     (POLLOUT | POLLWRBAND)
6414     #endif
6415     #define MY_PROVIDER    1 /* transport provider id */
6416     #define MAXSIZE        4000 /* size of send/receive buffer */
6417     #define TPLEN          30 /* maximum length of provider name */
6418     #define MAXCNX         10 /* maximum number of connections */

6419     extern int      errno;

6420     /*
6421     * Declaration of non-integer external functions
6422     */
6423     void      exit();
6424     void      perror();

6425     /* ===== */
6426     main()
6427     {

6428         register int      i; /* loop variable */
6429         register int      num; /* return value of t_snd() */
6430                                /* and t_rcv() */
6431         int                discflag = 0; /* flag to indicate a */
6432                                /* disc indication */
6433         int                errflag = 0; /* flag to indicate an error */
6434         int                event; /* stores events returned */
6435                                /* by t_look() */
6436         int                fd; /* current file descriptor */
6437         int                fdd; /* file descriptor */
6438                                /* for t_accept() */
6439         int                flags; /* used with t_rcv() */
6440         char                *datbuf; /* current send/receive buffer */
6441         unsigned int        act = 0; /* active endpoints */
6442         struct t_info        info; /* used with t_open() */
6443         struct t_bind        *preq; /* used with t_bind() */
6444         struct t_call        *pcall; /* used with t_listen() */
6445                                /* and t_accept() */
6446         struct t_discon      discon; /* used with t_rcvdis() */
6447         char                tpname[TPLEN]; /* transport provider name */
6448         char                buf[MAXCNX][MAXSIZE]; /* send/receive buffers */
6449         int                rcvdata[MAXCNX]; /* amount of data */
6450                                /* already received */
6451         int                snddata[MAXCNX]; /* amount of data already sent */

```

```

6452      struct pollfd      fds[MAXCNX];          /* used with poll() */

6453
6454      /*
6455      * Get name of transport provider
6456      */
6457      if (get_provider(MY_PROVIDER, tpname) == -1) {
6458          perror(">>> get_provider failed");
6459          exit(1);
6460      }
6461
6462      /*
6463      * Establish a transport endpoint in asynchronous mode
6464      */
6465      if ((fd = t_open(tpname, O_RDWR | O_NONBLOCK, &info)) == -1) {
6466          t_error(">>> t_open failed");
6467          exit(1);
6468      }
6469
6470      /*
6471      * Allocate memory for the parameters passed with t_bind().
6472      */
6473      if ((preq = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR)) == NULL) {
6474          t_error(">>> t_alloc(T_BIND) failed");
6475          t_close(fd);
6476          exit(1);
6477      }
6478
6479      /*
6480      * Given a symbolic name ("MY_NAME"), get_address returns an address
6481      * and its length in preq->addr.buf and preq->addr.len.
6482      */
6483      if (get_address("MY_NAME", &(preq->addr)) == -1) {
6484          perror(">>> get_address failed");
6485          t_close(fd);
6486          exit(1);
6487      }
6488      preq->qlen = 1;          /* is a listening endpoint */
6489
6490      /*
6491      * Bind the local protocol address to the transport endpoint.
6492      * The returned information is discarded.
6493      */
6494      if (t_bind(fd, preq, NULL) == -1) {
6495          t_error(">>> t_bind failed");
6496          t_close(fd);
6497          exit(1);
6498      }
6499
6500      /*
6501      * Allocate memory for the parameters used with t_listen.
6502      */
6503      if ((pcall = (struct t_call *) t_alloc(fd, T_CALL, T_ALL)) == NULL) {

```

```

6503         t_error(">>> t_alloc(T_CALL) failed");
6504         t_close(fd);
6505         exit(1);
6506     }
6507     /*
6508     * Initialise entry 0 of the fds array to the listening endpoint.
6509     * To be portable across different XTI implementations,
6510     * register for INEVENTS and not for POLLIN.
6511     */
6512     fds[act].fd = fd;
6513     fds[act].events = INEVENTS;
6514     fds[act].revents = 0;
6515     rcvdata[act] = 0;
6516     snddata[act] = 0;
6517     act = 1;
6518     /*
6519     * Enter an endless loop to wait for all incoming events.
6520     * Connect requests are accepted on new opened endpoints.
6521     * The example assumes that data is first sent by the client.
6522     * Then, the received data is sent back again and so on, until
6523     * the client disconnects.
6524     * Note that the total number of active endpoints (act) should
6525     * at least be 1, corresponding to the listening endpoint.
6526     */
6527     fprintf(stderr, "Waiting for XTI events...\n");
6528     while (act > 0) {
6529         /*
6530         * Wait for any events
6531         *
6532         */
6533         if (poll(&fds, (size_t)act, (int) -1) == -1) {
6534             perror(">>> poll failed");
6535             exit(1);
6536         }
6537         /*
6538         * Process incoming events on all active endpoints
6539         */
6540         for (i = 0 ; i < act ; i++) {
6541             if (fds[i].revents == 0)
6542                 continue; /* no event for this endpoint */
6543             if (fds[i].revents & ERREVENTS) {
6544                 fprintf(stderr, "[%d] Unexpected poll events: 0x%x\n",
6545                     fds[i].fd, fds[i].revents);
6546                 continue;
6547             }
6548             /*
6549             * set the current endpoint
6550             * set the current send/receive buffer
6551             */
6552             fd = fds[i].fd;
6553             datbuf = buf[i];
6554             /*
6555             * Check for events
6556             */
6557             switch((event = t_look(fd))) {
6558             case T_LISTEN:

```

```

6559      /*
6560      * Must be a connect indication
6561      */
6562      if (t_listen(fd, pcall) == -1) {
6563          t_error(">>> t_listen failed");
6564          exit(1);
6565      }
6566      /*
6567      * If it will exceed the maximum number
6568      * of connections that the server can handle,
6569      * reject the connect indication.
6570      */
6571      if (act >= MAXCNX) {
6572          fprintf(stderr, ">>> Connection request rejected\n");
6573          if (t_snddis(fd, pcall) == -1)
6574              t_error(">>> t_snddis failed");
6575          continue;
6576      }
6577      /*
6578      * Establish a transport endpoint
6579      * in asynchronous mode
6580      */
6581      if ((fdd = t_open(tpname, O_RDWR | O_NONBLOCK, &info))
6582          == -1) {
6583          t_error(">>> t_open failed");
6584          continue;
6585      }
6586      /*
6587      * Accept connection on this endpoint.
6588      * fdd no longer needs to be bound,
6589      * t_accept() will do it.
6590      */
6591      if (t_accept(fd, fdd, pcall) == -1) {
6592          t_error(">>> t_accept failed");
6593          t_close(fdd);
6594          continue;
6595      }
6596      fprintf(stderr, "Connection [%d] opened\n", fdd);
6597      /*
6598      * Register for all flags that might indicate
6599      * a T_DATA or T_DISCONNECT event, i. e.,
6600      * register for INEVENTS (to be portable
6601      * through all XTI implementations).
6602      */
6603      fds[act].fd = fdd;
6604      fds[act].events = INEVENTS;
6605      fds[act].revents = 0;
6606      rcvdata[act] = 0;
6607      snddata[act] = 0;
6608      act++;
6609      break;

6610      case T_DATA:
6611          /*
6612          * Must be a data indication
6613          */
6614          if ((num = t_rcv(fd, (datbuf + rcvdata[i]),
6615              (MAXSIZE - rcvdata[i]), &flags)) == -1) {

```

```

6616         switch (t_errno) {
6617         case TNODATA:
6618             /* No data is currently
6619              * available: repeat the loop
6620              */
6621             continue;
6622         case TLOOK:
6623             /* Must be a T_DISCONNECT event:
6624              * set discflag
6625              */
6626             event = t_look(fd);
6627             if (event == T_DISCONNECT) {
6628                 discflag = 1;
6629                 break;
6630             }
6631             else
6632                 fprintf(stderr, "Unexpected event %d\n",
6633                          event);
6634         default:
6635             /* Unexpected failure */
6636             t_error(">>> t_rcv failed");
6637             fprintf(stderr, "connection id: [%d]\n", fd);
6638             errflag = 1;
6639             break;
6640         }
6641     }
6642     if (discflag || errflag)
6643         /* exit from the event switch */
6644         break;
6645     fprintf(stderr, "[%d] %d bytes received\n", fd, num);
6646     rcvdata[i] += num;
6647     if (rcvdata[i] < MAXSIZE)
6648         continue;
6649     if (flags & T_MORE) {
6650         fprintf(stderr, "[%d] TSDU too long for receive
6651                  buffer\n", fd);
6652         errflag = 1;
6653         break; /* exit from the event switch */
6654     }
6655     /*
6656     * Send the data back:
6657     * Repeat t_snd() until either the whole TSDU
6658     * is sent back, or an event occurs.
6659     */
6660     fprintf(stderr, "[%d] sending data back\n", fd);
6661     do {
6662         if ((num = t_snd(fd, (datbuf + snddata[i]),
6663                        (MAXSIZE - snddata[i]), 0)) == -1) {
6664             switch (t_errno) {
6665             case TFLOW:
6666                 /*
6667                  * Register for the flags
6668                  * OUTEVENTS to get awakened by
6669                  * T_GODATA, and for INEVENTS
6670                  * to get aware of T_DISCONNECT
6671                  * or T_DATA.
6672                  */

```

```

6673         fds[i].events |= OUTEVENTS;
6674         continue;

6675         case TLOOK:
6676             /*
6677              * Must be a T_DISCONNECT event:
6678              * set discflag
6679              */
6680             event = t_look(fd);
6681             if (event == T_DISCONNECT) {
6682                 discflag = 1;
6683                 break;
6684             }
6685             else
6686                 fprintf(stderr, "Unexpected event %d\n",
6687                         event);

6688         default:
6689             t_error(">>> t_snd failed");
6690             fprintf(stderr, "connection id: [%d]\n", fd);
6691             errflag = 1;
6692             break;
6693     }
6694 }
6695 else {
6696     snddata[i] += num;
6697 }
6698 } while (MAXSIZE > snddata[i] && !discflag && !errflag);
6699 /*
6700  * Reset send/receive counters
6701  */
6702 rcvdata[i] = 0;
6703 snddata[i] = 0;
6704 break;

6705 case T_GODATA:
6706     /*
6707      * Flow control restriction has been lifted
6708      * restore initial event flags
6709      */
6710     fds[i].events = INEVENTS;
6711     continue;
6712 case T_DISCONNECT:
6713     /*
6714      * Must be a disconnect indication
6715      */
6716     discflag = 1;
6717     break;
6718 case -1:
6719     /*
6720      * Must be an error
6721      */
6722     t_error(">>> t_look failed");
6723     errflag = 1;
6724     break;
6725 default:
6726     /*
6727      * Must be an unexpected event
6728      */

```



```

6729         fprintf(stderr, "[%d] Unexpected event %d\n", fd, event);
6730         errflag = 1;
6731         break;
6732     }          /* end event switch */

6733     if (discflag) {
6734         /*
6735          * T_DISCONNECT has been received.
6736          * User data is not expected.
6737          */
6738         if (t_rcvdis(fd, &discon) == -1)
6739             t_error(">>> t_rcvdis failed");
6740         else
6741             fprintf(stderr, "[%d] Disconnect reason: 0x%x\n",
6742                    fd, discon.reason);
6743     }

6744     if (discflag || errflag) {
6745         /*
6746          * Close transport endpoint and
6747          * decrement number of active connections
6748          */
6749         t_close(fd);
6750         act--;
6751         /* Move last entry of fds array to current slot,
6752          * adjust internal counters and flags
6753          */
6754         fds[i].events = fds[act].events;
6755         fds[i].revents = fds[act].revents;
6756         fds[i].fd = fds[act].fd;
6757         discflag = 0; /* clear disconnect flag */
6758         errflag = 0; /* clear error flag */
6759         i--; /* Redo the for() event loop to consider
6760              * events related to the last entry of
6761              * fds array */
6762         fprintf(stderr, "Connection [%d] closed\n", fd);
6763     }

6764     }          /* end of for() event loop */

6765     }          /* end of while() loop */
6766     fprintf(stderr, ">>> Warning: no more active endpoints\n");
6767     exit(1);
6768 }

```

6769 C.8 The Select Function

6770 *select()* is defined in the 4.3 Berkeley Software Distribution as follows. Note that this definition
6771 may vary slightly in other systems.

6772 UX If the implementation defines `_XOPEN_UNIX`, refer to the description of *select()* in the **XSH**
6773 specification. Moreover, Chapter 8 on page 105 of the current document gives additional
6774 information on the specific effect of *select()* when applied to Sockets.

6775 The manual page for this definition is given on the next page, and this is followed by a section
6776 giving guidelines for Use of BSD *select()*.

6777 **NAME**

6778 select - synchronous I/O multiplexing

6779 **SYNOPSIS**

6780 #include <sys/types.h>

6781 #include <sys/time.h>

6782 nfound = select(nfds, readfds, writefds, exceptfds, timeout)

6783 int nfound, nfds;

6784 fd_set *readfds, *writefds, *exceptfds;

6785 struct timeval *timeout;

6786 FD_SET(fd, &fdset)

6787 FD_CLR(fd, &fdset)

6788 FD_ISSET(fd, &fdset)

6789 FD_ZERO(&fdset)

6790 int fd;

6791 fd_set fdset;

6792 **DESCRIPTION**

6793 *select()* examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds* and
 6794 *exceptfds* to see if some of their descriptors are ready for reading, ready for writing, or have an
 6795 exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set;
 6796 that is, the descriptors from 0 through *nfds* -1 in the descriptor sets are examined. On return,
 6797 *select()* replaces the given descriptor sets with subsets consisting of those descriptors that are
 6798 ready for the requested operation. The total number of ready descriptors in all the sets is
 6799 returned in *nfound*.

6800 The descriptor sets are stored as bit fields in arrays of integers. The following macros are
 6801 provided for manipulating such descriptor sets: *FD_ZERO(&fdset)* initialises a descriptor set
 6802 *fdset* to the null set. *FD_SET(fd, &fdset)* includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd,*
 6803 *&fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, &fdset)* is non-zero if *fd* is a member of *fdset*, zero
 6804 otherwise. The behaviour of these macros is undefined if a descriptor value is less than zero or
 6805 greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number
 6806 of descriptors supported by the system.

6807 If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to
 6808 complete. If *timeout* is a zero pointer, the select blocks indefinitely. To affect a poll, the *timeout*
 6809 argument should be non-zero, pointing to a zero-valued *timeval* structure.

6810 Any of *readfds*, *writefds* and *exceptfds* may be given as zero pointers if no descriptors are of
 6811 interest.

6812 **RETURN VALUES**

6813 *select()* returns the number of ready descriptors that are contained in the descriptor sets, or -1 if
 6814 an error occurred. If the time limit expires then *select()* returns 0. If *select()* returns with an
 6815 error, including one due to an interrupted call, the descriptor sets will be unmodified.

6816 **ERRORS**6817 An error return from *select()* indicates:

6818 [EBADF] One of the descriptor sets specified an invalid descriptor.

select()

6819	[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
6820		
6821	[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.

6822 C.9 Use of Select

6823 Many systems provide the macros *FD_SET*, *FD_CLR*, *FD_ISSET* and *FD_ZERO* in `<sys/types.h>`
6824 or other header files to manipulate these bit masks. If not available they should be defined by
6825 the user (see the program example below).

6826 For an application to be notified of any XTI events on each of its active endpoints identified by a
6827 file descriptor *fd*, this file descriptor *fd* should be included in the appropriate descriptor sets
6828 *readfds*, *exceptfds* or *writefds* as specified below:

6829 • For Class 1 events:

6830 Set the bit masks *readfds* and *exceptfds* by *FD_SET(fd, readfds)* and *FD_SET(fd, exceptfds)*.

6831 • For Class 2 events:

6832 Set the bit mask *writefds* by *FD_SET(fd, writefds)*.

6833 If, on return of *select()*, the bit corresponding to *fd* is set in *writefds*, this can have a different
6834 meaning for different XTI implementations. It could either mean:

6835 • that both normal and expedited data may be sent

6836 or:

6837 • that normal data may be sent and the flow of expedited data cannot be monitored via *select()*.

6838 A truly portable XTI application should, therefore, not assume that the flow of expedited data is
6839 monitored by *select()*. This is not a serious restriction, since an application usually only sends
6840 small amounts of expedited data and flow restrictions are not a major problem.

6841 The remainder of this section describes the outline of an XTI server program making use of the
6842 BSD *select()*.

```

6843  /*
6844  * This is a simple server application example to show how select() can
6845  * be used in a portable manner to wait for the occurrence of XTI events.
6846  * In this example, select() is used to wait for the events T_LISTEN,
6847  * T_DISCONNECT, T_DATA and T_GODATA.
6848  *
6849  * A transport endpoint is opened in asynchronous mode over a
6850  * message-oriented transport provider (for example, ISO). The endpoint is
6851  * bound with qlen = 1, and the application enters an endless loop to wait
6852  * for all incoming XTI events on all its active endpoints.
6853  * For all connect indications received, a new endpoint is opened with
6854  * qlen = 0 and the connect request is accepted on that endpoint.
6855  * For all established connections, the application waits for data to be
6856  * received from one of its clients, sends the received data back to the
6857  * sender and waits for data again.
6858  * The cycle repeats until all the connections are released by the clients.
6859  * The disconnect indications are processed and the endpoints closed.
6860  *
6861  * The example references two fictitious functions:
6862  *
6863  * - int get_provider(int tpid, char * tpname)
6864  *   Given a number as transport provider id, the function returns in
6865  *   tpname a string as transport provider name that can be used with
6866  *   t_open(). This function hides the different naming schemes of
6867  *   different XTI implementations.
6868  *
6869  * - int get_address(char * symb_name, struct netbuf address)
6870  *   Given a symbolic name symb_name and a pointer to a struct netbuf
6871  *   with allocated buffer space as input, the function returns a
6872  *   protocol address. This function hides the different addressing
6873  *   schemes of different XTI implementations.
6874  */
6875  /*
6876  * General Includes
6877  */
6878  #include <fcntl.h>
6879  #include <stdio.h>
6880  #include <xti.h>
6881  /*
6882  * Include files for select(). Some UNIX derivatives use other includes,
6883  * for example, <sys/times.h> instead of <sys/time.h>.
6884  *   <sys/select.h> instead of <sys/types.h>.
6885  */
6886  #include <sys/types.h>
6887  #include <time.h>
6888  /*
6889  * Includes that are only relevant, if the type fd_set and the macros
6890  * FD_SET, FD_CLR, FD_ISSET and FD_ZERO have to be explicitly defined
6891  * in this program.
6892  */
6893  #include <limits.h>
6894  #include <string.h>      /* for memset() */
6895  /*
6896  * Various Defines
6897  */

```

```

6898     #define MY_PROVIDER      1    /* transport provider id */
6899     #define MAXSIZE          4000  /* size of send/receive buffer */
6900     #define TPLEN            30    /* maximum length of provider name */
6901     #define MAXCNX           10    /* maximum number of connections */

6902     /*
6903     * Select uses bit masks of file descriptors in longs. Most systems
6904     * provide a type "fd_set" and macros in <sys/types.h> or <sys/select.h>
6905     * to ease the use of select().
6906     * They are explicitly defined below in case that they are not defined in
6907     * <sys/types.h> or <sys/select.h>.
6908     */
6909     /*
6910     * OPEN_MAX should be >= number of maximum open files per process
6911     */
6912     #ifndef OPEN_MAX
6913     #define OPEN_MAX          256
6914     #endif
6915     #ifndef NFDBITS
6916     #define NFDBITS (sizeof(long) * CHAR_BIT)    /* bits per mask */
6917     #endif
6918     #ifndef howmany
6919     #define howmany(x, y)    (((x)+(y)-1)/(y))
6920     #endif
6921     #ifndef FD_SET
6922     typedef struct fd_set {
6923         long    fds_bits[howmany(OPEN_MAX, NFDBITS)];
6924     } fd_set;
6925     #define FD_SET(n, p)    ((p)->fds_bits[(n)/NFDBITS] |= (1 << ((n) % NFDBITS)))
6926     #define FD_CLR(n, p)    ((p)->fds_bits[(n)/NFDBITS] &= ~(1 << ((n) % NFDBITS)))
6927     #define FD_ISSET(n, p) ((p)->fds_bits[(n)/NFDBITS] & (1 << ((n) % NFDBITS)))
6928     #define FD_ZERO(p)    memset(*(p), (u_char) 0, sizeof(*(p)))
6929     #endif /* ! FD_SET */

6930     extern int    errno;

6931     /*
6932     * Declaration of non-integer external functions.
6933     */
6934     void    exit();
6935     void    perror();

6936     /* ===== */

6937     main()
6938     {

6939         register int    i;                /* loop variable */
6940         register int    num;              /* return value of t_snd() */
6941                                         /* and t_rcv() */

6942         int              discflag = 0;    /* flag to indicate a */
6943                                         /* disc indication */
6944         int              errflag = 0;    /* flag to indicate an error */
6945         int              event;          /* stores events returned */

```

```

6946                                     /* by t_look() */
6947     int          fd;                  /* current file descriptor */
6948     int          fdd;                 /* file descriptor */
6949                                     /* for t_accept() */
6950     int          flags;               /* used with t_rcv() */
6951     char         *datbuf;             /* current send/receive */
6952                                     /* buffer */
6953     size_t       act = 0;             /* active endpoints */
6954     struct t_info info;               /* used with t_open() */
6955     struct t_bind *preq;              /* used with t_bind() */
6956     struct t_call *pcall;             /* used with t_listen() */
6957                                     /* and t_accept() */
6958     struct t_discon discon;           /* used with t_rcvdis() */
6959     char         tpname[TPLEN];      /* transport provider name */

6960     int          fds[MAXCNX];         /* array of file descriptors */
6961     char         buf[MAXCNX][MAXSIZE] /* send/receive buffers */
6962     int          rcvdata[MAXCNX];    /* amount of data */
6963                                     /* already received */
6964     int          snddata[MAXCNX];    /* amount of data already sent */

6965     fd_set rfdsets, wfds, xfds;      /* file descriptor sets */
6966                                     /* for select() */
6967     fd_set rfdds, wfdds, xfdds;      /* initial values of */
6968                                     /* file descriptor sets */
6969                                     /* rfdsets, wfds and xfds */

6970     /*
6971     * Get name of transport provider
6972     */
6973     if (get_provider(MY_PROVIDER, tpname) == -1) {
6974         perror(">>> get_provider failed");
6975         exit(1);
6976     }

6977     /*
6978     * Establish a transport endpoint in asynchronous mode
6979     */
6980     if ((fd = t_open(tpname, O_RDWR | O_NONBLOCK, &info)) == -1) {
6981         t_error(">>> t_open failed");
6982         exit(1);
6983     }

6984     /*
6985     * Allocate memory for the parameters passed with t_bind().
6986     */
6987     if ((preq = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR)) == NULL) {
6988         t_error(">>> t_alloc(T_BIND) failed");
6989         t_close(fd);
6990         exit(1);
6991     }

6992     /*
6993     * Given a symbolic name ("MY_NAME"), get_address returns an address
6994     * and its length in preq->addr.buf and preq->addr.len.
6995     */
6996     if (get_address("MY_NAME", &(preq->addr)) == -1) {
6997         perror(">>> get_address failed");
6998         t_close(fd);

```



```

6999         exit(1);
7000     }
7001     preq->qlen = 1;          /* is a listening endpoint */
7002     /*
7003     * Bind the local protocol address to the transport endpoint.
7004     * The returned information is discarded.
7005     */
7006     if (t_bind(fd, preq, NULL) == -1) {
7007         t_error(">>> t_bind failed");
7008         t_close(fd);
7009         exit(1);
7010     }
7011     if (t_free(preq, T_BIND) == -1) {
7012         t_error(">>> t_free failed");
7013         t_close(fd);
7014         exit(1);
7015     }
7016     /*
7017     * Allocate memory for the parameters used with t_listen.
7018     */
7019     if ((pcall = (struct t_call *) t_alloc(fd, T_CALL, T_ALL)) == NULL) {
7020         t_error(">>> t_alloc(T_CALL) failed");
7021         t_close(fd);
7022         exit(1);
7023     }
7024     /*
7025     * Initialise listening endpoint in descriptor set.
7026     * To be portable across different XTI implementations,
7027     * register for descriptor set rfdds and xfdds
7028     */
7029     FD_ZERO(&rfdds);
7030     FD_ZERO(&xfdds);
7031     FD_ZERO(&wfdds);
7032     FD_SET(fd, &rfdds);
7033     FD_SET(fd, &xfdds);
7034     fds[act] = fd;
7035     rcvdata[act] = 0;
7036     snddata[act] = 0;
7037     act = 1;
7038     /*
7039     * Enter an endless loop to wait for all incoming events.
7040     * Connect requests are accepted on a new opened endpoint.
7041     * The example assumes that data is first sent by the client.
7042     * Then, the received data is sent back again and so on, until
7043     * the client disconnects.
7044     * Note that the total number of active endpoints (act) should
7045     * at least be 1, corresponding to the listening endpoint.
7046     */
7047     fprintf(stderr, "Waiting for XTI events...\n");
7048     while (act > 0) {
7049         /*
7050         * Wait for any events
7051         */
7052     }

```

```

7053     * Set the mask sets rfd, xfd and wfd to their initial values
7054     */
7055     rfd = rfd;
7056     xfd = xfd;
7057     wfd = wfd;
7058     if (select(OPEN_MAX, &rfd, &wfd, &xfd,
7059             (struct timeval *) NULL) == -1) {
7060         perror(">>> select failed");
7061         exit(1);
7062     }
7063     /*
7064     * Process incoming events on all active endpoints
7065     */
7066     for (i = 0 ; i < act ; i++) {
7067         /*
7068         * set the current endpoint
7069         * set the current send/receive buffer
7070         */
7071         fd = fds[i];
7072         datbuf = buf[i];

7073         if (FD_ISSET(fd, &xfd)) {
7074             fprintf(stderr, "[%d] Unexpected select events\n", fd);
7075             continue;
7076         }
7077         if (!FD_ISSET(fd, &rfd) && !FD_ISSET(fd, &wfd))
7078             continue; /* no event for this endpoint */

7079         /*
7080         * Check for events
7081         */
7082         switch((event = t_look(fd))) {
7083         case T_LISTEN:
7084             /*
7085             * Must be a connect indication
7086             */
7087             if (t_listen(fd, pcall) == -1) {
7088                 t_error(">>> t_listen failed");
7089                 exit(1);
7090             }

7091             /*
7092             * If it will exceed the maximum number
7093             * of connections that the server can handle,
7094             * reject the connect indication.
7095             */
7096             if (act >= MAXCNX) {
7097                 fprintf(stderr, ">>> Connection request
7098                         rejected\n");
7099                 if (t_snddis(fd, pcall) == -1)
7100                     t_error(">>> t_snddis failed");
7101                 continue;
7102             }
7103             /*
7104             * Establish a transport endpoint
7105             * in asynchronous mode
7106             */
7107             if ((fdd = t_open(tpname, O_RDWR | O_NONBLOCK,
7108                             &info)) == -1) {

```

```

7109         t_error(">>> t_open failed");
7110         continue;
7111     }
7112     /*
7113     * Accept connection on this endpoint.
7114     * fdd no longer needs to be bound,
7115     * t_accept() will do it
7116     */
7117     if (t_accept(fd, fdd, pcall) == -1) {
7118         t_error(">>> t_accept failed");
7119         t_close(fdd);
7120         continue;
7121     }
7122     fprintf(stderr, "Connection [%d] opened\n", fdd);
7123
7124     /*
7125     * Register for all flags that might indicate
7126     * a T_DATA or T_DISCONNECT event, i. e.,
7127     * register for rfdds and xfdds (to be portable
7128     * through all XTI implementations).
7129     */
7129     fds[act] = fdd;
7130     FD_SET(fdd, &rfdds);
7131     FD_SET(fdd, &xfdds);
7132     rcvdata[act] = 0;
7133     snddata[act] = 0;
7134     act++;
7135     break;
7136
7136     case T_DATA:
7137         /* Must be a data indication
7138         */
7139         if ((num = t_rcv(fd, (datbuf + rcvdata[i]),
7140             (MAXSIZE - rcvdata[i]), &flags)) == -1) {
7141             switch (t_errno) {
7142                 case TNODATA:
7143                     /* No data is currently
7144                     * available: repeat the loop
7145                     */
7146                     continue;
7147                 case TLOOK:
7148                     /* Must be a T_DISCONNECT event:
7149                     * set discflag
7150                     */
7151                     event = t_look(fd);
7152                     if (event == T_DISCONNECT) {
7153                         discflag = 1;
7154                         break;
7155                     }
7156                     else
7157                         fprintf(stderr, "Unexpected event %d\n", event);
7158
7158                 default:
7159                     /* Unexpected failure */
7160                     t_error(">>> t_rcv failed");
7161                     fprintf(stderr, "connection id: [%d]\n", fd);
7162                     errflag = 1;
7163                     break;
7164             }

```

```

7165     }
7166     if (discflag || errflag)
7167         /* exit from the event switch */
7168         break;
7169     fprintf(stderr, "[%d] %d bytes received\n", fd, num);
7170     rcvdata[i] += num;
7171     if (rcvdata[i] < MAXSIZE)
7172         continue;
7173     if (flags & T_MORE) {
7174         fprintf(stderr, "[%d] TSDU too long for receive
7175                 buffer\n", fd);
7176         errflag = 1;
7177         break; /* exit from the event switch */
7178     }
7179     /*
7180     * Send the data back.
7181     * Repeat t_snd() until either the whole TSDU
7182     * is sent back, or an event occurs.
7183     */
7184     fprintf(stderr, "[%d] sending data back\n", fd);
7185     do {
7186         if ((num = t_snd(fd, (datbuf + snddata[i]),
7187             (MAXSIZE - snddata[i]), 0)) == -1) {
7188             switch (t_errno) {
7189             case TFLOW:
7190                 /*
7191                 * Register for wfds to get
7192                 * awoken by T_GODATA, and for
7193                 * rfds and xfds to get aware of
7194                 * T_DISCONNECT or T_DATA.
7195                 */
7196                 FD_SET(fd, &wfdds);
7197                 continue;
7198             case TLOOK:
7199                 /*
7200                 * Must be a T_DISCONNECT event:
7201                 * set discflag
7202                 */
7203                 event = t_look(fd);
7204                 if (event == T_DISCONNECT) {
7205                     discflag = 1;
7206                     break;
7207                 }
7208                 else
7209                     fprintf(stderr, "Unexpected event
7210                             %d\n", event);
7211             default:
7212                 t_error(">>> t_snd failed");
7213                 fprintf(stderr, "connection id: [%d]\n", fd);
7214                 errflag = 1;
7215                 break;
7216             }
7217         }
7218     } else {
7219         snddata[i] += num;

```

```

7220         }
7221     } while (MAXSIZE > snddata[i] && !discflag && !errflag);
7222     /*
7223     * Reset send/receive counter
7224     */
7225     rcvdata[i] = 0;
7226     snddata[i] = 0;
7227     break;

7228     case T_GODATA:
7229     /*
7230     * Flow control restriction has been lifted
7231     * restore initial event flags
7232     */
7233     FD_CLR(fd, &wfdds);
7234     continue;
7235     case T_DISCONNECT:
7236     /*
7237     * Must be a disconnect indication
7238     */
7239     discflag = 1;
7240     break;
7241     case -1:
7242     /*
7243     * Must be an error
7244     */
7245     t_error(">>> t_look failed");
7246     errflag = 1;
7247     break;
7248     default:
7249     /*
7250     * Must be an unexpected event
7251     */
7252     fprintf(stderr, "[%d] Unexpected event %d\n", fd, event);
7253     errflag = 1;
7254     break;
7255     } /* end event switch */

7256     if (discflag) {
7257     /*
7258     * T_DISCONNECT has been received.
7259     * User data is not expected.
7260     */
7261     if (t_rcvdis(fd, &discon) == -1)
7262         t_error(">>> t_rcvdis failed");
7263     else
7264         fprintf(stderr, "[%d] Disconnect reason: 0x%x\n",
7265                 fd, discon.reason);
7266     }

7267     if (discflag || errflag) {
7268     /*
7269     * Close transport endpoint and
7270     * decrement number of active connections
7271     */
7272     t_close(fd);
7273     act--;
7274     /*
7275     * Unregister fd from initial mask sets

```

```
7276         */
7277         FD_CLR(fd, &rfdds);
7278         FD_CLR(fd, &xfdds);
7279         FD_CLR(fd, &wfdds);
7280         /* Move last entry of fds array to current slot,
7281          * adjust internal counters and flags
7282          */
7283         fds[i] = fds[act];
7284         discflag = 0; /* clear disconnect flag */
7285         errflag = 0; /* clear error flag */
7286         i--; /* Redo the for() event loop to consider
7287              * events related to the last entry of
7288              * fds array */
7289         fprintf(stderr, "Connection [%d] closed\n", fd);
7290     }
7291 } /* end of for() event loop */
7292 } /* end of while() loop */
7293 fprintf(stderr, ">>> Warning: no more active endpoints\n");
7294 exit(1);
7295 }
```

Use of XTI to Access NetBIOS

7296

7297 D.1 Introduction

7298 NetBIOS represents an important *de facto* standard for networking DOS and OS/2 PCs. The
7299 X/Open Specification **Protocols for X/Open PC Interworking: SMB** (see the referenced
7300 **NetBIOS** specification) provides mappings of NetBIOS services to OSI and IPS transport
7301 protocols.⁷

7302 The following CAE Specification extends that work to provide a standard programming
7303 interface to NetBIOS transport providers in X/Open-compliant systems, using an existing
7304 X/Open Common Applications Environment (CAE) interface, XTI.

7305 The X/Open Transport Interface (XTI) defines a transport service interface that is independent of
7306 any specific transport provider.

7307 This CAE Specification defines a standard for using XTI to access NetBIOS transport providers.
7308 Applications that use XTI to access NetBIOS transport providers are referred to as “transport
7309 users”.

7310 D.2 Objectives

7311 The objectives of this standardisation are:

- 7312 1. to facilitate the development and portability of CAE applications that interwork with the
7313 large installed base of NetBIOS applications in a Local Area Network (LAN) environment;
- 7314 2. to enable a single application to use the same XTI interface to communicate with remote
7315 applications through either an IPS profile, an OSI profile or a NetBIOS profile (that is, RFC
7316 1001/1002 or TOP/NetBIOS),
- 7317 3. to provide a common interface that can be used for IPC with clients using either (PC)NFS
7318 or SMB protocols for resources sharing.

7319 This CAE Specification provides a migration step to users moving from proprietary systems in a
7320 NetBIOS environment to open systems, that is, the X/Open CAE.

7321

7322 7. The mappings are defined by the Specification of NetBIOS Interface and Name Service Support by Lower Layer OSI Protocols,
7323 and RFC-1001/RFC-1002 respectively. See the referenced **NetBIOS** specification. The relevant chapters are Chapter 13,
7324 NetBIOS Interface to ISO Transport Services, Chapter 14, Protocol Standard for a NetBIOS Service on a TCP/UDP Transport:
7325 Concepts and Methods and Chapter 15, Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Detailed
7326 Specification.

7327 D.3 Scope

7328 No extensions to XTI, as it is defined in the main body of this CAE Specification, are made in this
7329 NetBIOS CAE Specification. This NetBIOS CAE Specification is concerned only with
7330 standardisation of the mapping of XTI to the NetBIOS facilities, and not a new definition of XTI
7331 itself.

7332 This CAE Specification applies only to the use of XTI in the single NetBIOS subnetwork case,
7333 and does not provide for the support of applications operating in multiple, non-overlapping
7334 NetBIOS name spaces.

7335 The following NetBIOS facilities found in various NetBIOS implementations are considered
7336 outside the scope of XTI (note that this list is not necessarily definitive):

- 7337 • LAN.STATUS.ALERT
- 7338 • RESET
- 7339 • SESSION STATUS
- 7340 • TRACE
- 7341 • UNLINK
- 7342 • RPL (Remote Program Load)
- 7343 • ADAPTER STATUS
- 7344 • FIND NAME
- 7345 • SEND.NOACK
- 7346 • CHAIN.SEND.NOACK
- 7347 • CANCEL
- 7348 • receiving a datagram on any name
- 7349 • receiving data on any connection.

7350 It must also be noted that not all commands are specified in the protocols.

7351 Omitting these does not restrict interoperability with the majority of NetBIOS implementations,
7352 since they have local significance only (RESET, SESSION STATUS), are concerned with systems
7353 management (UNLINK, RPL, ADAPTER STATUS), or are LAN- and vendor-specific (FIND
7354 NAME). If and how these functions are made available to the programmer is left to the
7355 implementor of this particular XTI implementation.

7356 **D.4 Issues**

7357 The primary issues for XTI as a transport interface to NetBIOS concern the passing of NetBIOS
7358 names and name type information through XTI, specification of restrictions on XTI functions in
7359 the NetBIOS environment, and handling the highly dynamic assignment of NetBIOS names.

7360 **D.5 NetBIOS Names and Addresses**

7361 NetBIOS uses 16-octet alphanumeric names as “transport” addresses. NetBIOS names must be
 7362 exactly 16 octets, with shorter names padded with spaces to 16 octets. In addition, NetBIOS
 7363 names are either unique names or group names, and must be identified as such in certain
 7364 circumstances.

7365 The following restrictions should be applied to NetBIOS names. Failure to observe these
 7366 restrictions may result in unpredictable results.

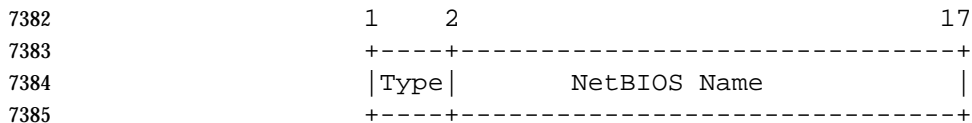
- 7367 1. Byte 0 of the name is not allowed to be hexadecimal 00 (0x00).
- 7368 2. Byte 0 of the name is not allowed to be an asterisk, except as noted elsewhere in this
 7369 specification to support broadcast datagrams.
- 7370 3. Names should not begin with company names or trademarks.
- 7371 4. Names should not begin with hexadecimal FF (0xFF).
- 7372 5. Byte 15 of the name should not be in the range 0x00 - 0x1F.

7373 The concept of a permanent node name, as provided in the native NetBIOS environment, is not
 7374 supported in the X/Open CAE.

7375 The following definitions are supplied with any implementation of XTI on top of NetBIOS. They
 7376 should be included in <xti.h>.

```
7377 #define NB_UNIQUE      0
7378 #define NB_GROUP      1
7379 #define NB_NAMELEN    16
7380 #define NB_BCAST_NAME " " /* asterisk plus 15 spaces */
```

7381 The protocol addresses passed in calls to *t_bind()*, *t_connect()*, etc., are structured as follows:



7386 **Type** The first octet specifies the type of the NetBIOS name. It may be set to
 7387 NB_UNIQUE or NB_GROUP.

7388 **NetBIOS Name** Octets 2 through 17 contain the 16-octet NetBIOS name.

7389 All NetBIOS names, complete with the name type identifier, are passed through XTI in a *netbuf*
 7390 address structure (that is, **struct netbuf addr**), where *addr.buf* points to a NetBIOS protocol
 7391 address as defined above. This applies to all XTI functions that pass or return a (NetBIOS)
 7392 protocol address (for example, *t_bind()*, *t_connect()*, *t_rcvudata()*, etc.).

7393 Note, however, that only the *t_bind()* and *t_getprotaddr()* functions use the name type
 7394 information. All other functions ignore it.

7395 If the NetBIOS protocol address is returned, the name type information is to be ignored since the
 7396 NetBIOS transport providers do not provide the type information in the connection
 7397 establishment phase.

7398 NetBIOS names can become invalid even after they have been registered successfully due to the
 7399 NetBIOS name conflict resolution process (for example, Top/NetBIOS NameConflictAdvise
 7400 indication). For existing NetBIOS connections this has no effect since the connection endpoint
 7401 can still be identified by the *fd*. However, in the connection establishment phase *2t_listen()* and
 7402 *t_connect()* this event is indicated by setting *t_errno* to [TBADF].

7403 D.6 NetBIOS Connection Release

7404 Native NetBIOS implementations provide a linger-on-close release mechanism whereby a
7405 transport disconnect request (NetBIOS HANGUP) will not complete until all outstanding send
7406 commands have completed. NetBIOS attempts to deliver all queued data by delaying, if
7407 necessary, disconnection for a period of time. The period of time might be configurable; a value
7408 of 20 seconds is common practice. Data still queued after this time period may get discarded so
7409 that delivery cannot be guaranteed.

7410 XTI, however, offers two different modes to release a connection: an abortive mode via
7411 *t_snddis()/t_rcvdis()*, and a graceful mode via *t_sndrel()/t_rcvrel()*. If a connection release is
7412 initiated by a *t_snddis()*, queued send data may be discarded. Only the use of *t_sndrel()*
7413 guarantees that the linger-on-close mechanism is enabled as described above. The support of
7414 *t_sndrel()/t_rcvrel()* is optional and only provided by implementations with servtype
7415 T_COTS_ORD (see *t_getinfo()* in Section D.8 on page 245).

7416 A call to *t_sndrel()* initiates the linger-on-close mechanism and immediately returns with the XTI
7417 state changed to T_OUTREL. The NetBIOS provider sends all outstanding data followed by a
7418 NetBIOS Close Request. After receipt of a NetBIOS Close Response, the NetBIOS provider
7419 informs the transport user, via the event T_ORDREL, that is to be consumed by calling *t_rcvrel()*.
7420 If a timeout occurs, however, a T_DISIN with a corresponding reason code is generated.

7421 Receive data arriving before the NetBIOS Close Request is sent is indicated by T_DATA and can
7422 be read by the transport user.

7423 Calling *t_snddis()* initiates an abortive connection release and immediately returns with the XTI
7424 state changed to T_IDLE. Outstanding send and receive data may be discarded. The NetBIOS
7425 provider sends as many outstanding data as possible prior to closing the connection, but
7426 discards any receive data. Some outstanding data may be discarded by the *t_snddis()*
7427 mechanism, so that not all data can be sent by the NetBIOS provider. Furthermore, an occurring
7428 timeout condition could not be indicated to the transport user.

7429 An incoming connection release will always result in a T_DISCONNECT event, never in a
7430 T_ORDREL event. To be precise, if the NetBIOS provider receives a Close Request, it discards
7431 any pending send and receive data, sends a Close Response and informs the transport user via
7432 T_DISCONNECT.

7433 **D.7 Options**

7434 No NetBIOS-specific options are defined. An implementation may, however, provide XTI-level
7435 options (see *t_optmgmt()* on page 76).

7436 **D.8 XTI Functions**

7437	<i>t_accept()</i>	No user data may be returned to the caller (call->udata.len=0).
7438		This function may only be used with connection-oriented transport endpoints.
7439		The <i>t_accept()</i> function will fail if a user attempts to accept a connection
7440		request on a connectionless endpoint and <i>t_errno</i> will be set to
7441		[TNOTSUPPORT].
7442	<i>t_alloc()</i>	No special considerations for NetBIOS transport providers.
7443	<i>t_bind()</i>	The NetBIOS name and name type values are passed to the transport provider
7444		in the <i>req</i> parameter (req->addr.buf) and the actual bound address is returned
7445		in the <i>ret</i> parameter (ret->addr.buf), as described earlier in Section D.5 on page
7446		242. If the NetBIOS transport provider is unable to register the name specified
7447		in the <i>req</i> parameter, the call to <i>t_bind()</i> will fail with <i>t_errno</i> set to
7448		[TADDRBUSY] if the name is already in use, or to [TBADADDR] if it was an
7449		illegal NetBIOS name.
7450		If the <i>req</i> parameter is a null pointer or req->addr.len=0, the transport provider
7451		may assign an address for the user. This may be useful for outgoing
7452		connections on which the name of the caller is not important.
7453		If the name specified in <i>req</i> parameter is NB_BCAST_NAME, <i>qlen</i> must be
7454		zero, and the transport endpoint the name is bound to is enabled to receive
7455		broadcast datagrams. In this case, the transport endpoint must support
7456		connectionless service, otherwise the <i>t_bind()</i> function will fail and <i>t_errno</i>
7457		will be set to [TBADADDR].
7458	<i>t_close()</i>	No special considerations for NetBIOS transport providers.
7459		It is assumed that the NetBIOS transport provider will release the NetBIOS
7460		name associated with the closed endpoint if this is the only endpoint bound to
7461		this name and the name has not already been released as the result of a
7462		previous <i>t_unbind()</i> call on this endpoint.
7463	<i>t_connect()</i>	The NetBIOS name of the destination transport user is provided in the <i>sndcall</i>
7464		parameter (sndcall->addr.buf), and the NetBIOS name of the responding
7465		transport user is returned in the <i>rcvcall</i> parameter (rcvcall->addr.buf), as
7466		described in Section D.5 on page 242. If the connection is successful, the
7467		NetBIOS name of the responding transport user will always be the same as
7468		that specified in the <i>sndcall</i> parameter.
7469		Local NetBIOS connections are supported. NetBIOS datagrams are sent, if
7470		applicable, to local names as well as remote names. No user data may be sent
7471		during connection establishment (udata.len=0 in <i>sndcall</i>).
7472		This function may only be used with connection-oriented transport endpoints.
7473		The <i>t_connect()</i> function will fail if a user attempts to initiate a connection on a
7474		connectionless endpoint and <i>t_errno</i> will be set to [TNOTSUPPORT].
7475		[TBADF] may be returned in the case that the NetBIOS name associated with
7476		the <i>fd</i> referenced in the <i>t_connect()</i> call is no longer in the CAE system name
7477		table (see Section D.5 on page 242).
7478	<i>t_error()</i>	No special considerations for NetBIOS transport providers.
7479	<i>t_free()</i>	No special considerations for NetBIOS transport providers.

7480	<i>t_getinfo()</i>	The values of the parameters in the <i>t_info</i> structure will reflect NetBIOS transport limitations, as follows:
7481		
7482		<i>addr</i> <i>sizeof()</i> the NetBIOS protocol address, as defined in Section D.5
7483		on page 242.
7484		<i>options</i> Equals -2, indicating no user-settable options.
7485		<i>tsdu</i> Equals the size returned by the transport provider. If the <i>fd</i> is
7486		associated with a connection-oriented endpoint it is a positive
7487		value, not larger than 131070. If the <i>fd</i> is associated with a
7488		connectionless endpoint it is a positive value not larger than
7489		65535 ⁸ .
7490		<i>etsdu</i> Equals -2, indicating expedited data is not supported.
7491		<i>connect</i> Equals -2, indicating data cannot be transferred during
7492		connection establishment.
7493		<i>discon</i> Equals -2, indicating data cannot be transferred during
7494		connection release.
7495		<i>servtype</i> Set to <i>T_COTS</i> if the <i>fd</i> is associated with a connection-oriented
7496		endpoint, or <i>T_CLTS</i> if associated with a connectionless
7497		endpoint. Optionally, may be set to <i>T_COTS_ORD</i> if the <i>fd</i> is
7498		associated with a connection-oriented endpoint and the
7499		transport provider supports the use of <i>t_sndrel()/t_rcvrel()</i> as
7500		described in Section D.6 on page 243.
7501		<i>flags</i> Equals <i>T_SNDZERO</i> , indicating that zero TSDUs may be sent.
7502	<i>t_getprotaddr()</i>	The NetBIOS name and name type of the transport endpoint referred to by the
7503		<i>fd</i> are passed in the <i>boundaddr</i> parameter (<i>boundaddr->addr.buf</i>), as described
7504		in Section D.5 on page 242; 0 is returned in <i>boundaddr->addr.len</i> if the
7505		transport endpoint is in the <i>T_UNBND</i> state. The NetBIOS name currently
7506		connected to <i>fd</i> , if any, is passed in the <i>peeraddr</i> parameter (<i>peeraddr-></i>
7507		<i>addr.buf</i>); the value 0 is returned in <i>peeraddr->addr.len</i> if the transport
7508		endpoint is not in the <i>T_DATAXFER</i> state.
7509	<i>t_getstate()</i>	No special considerations for NetBIOS transport providers.
7510	<i>t_listen()</i>	On return, the <i>call</i> parameter provides the NetBIOS name of the calling
7511		transport user (that issued the connection request), as described in Section D.5
7512		on page 242.
7513		No user data may be transferred during connection establishment (<i>call-></i>
7514		<i>udata.len=0</i> on return).
7515		This function may only be used with connection-oriented transport endpoints.
7516		The <i>t_listen()</i> function will fail if a user attempts to <i>listen</i> on a connectionless
7517		endpoint and <i>t_errno</i> will be set to [TNOTSUPPORT]. [TBADF] may be
7518		returned in the case that the NetBIOS name associated with the <i>fd</i> referenced
7519		in the <i>t_listen()</i> function is no longer in the CAE system name table, as may
7520		occur as a result of the NetBIOS name conflict resolution process (for example,
7521	_____	
7522	8.	For the mappings to OSI and IPS protocols, the value cannot exceed 512 or 1064 respectively.

7523		TOP/NetBIOS NameConflictAdvise indication).
7524	<i>t_look()</i>	Since expedited data is not supported in NetBIOS, the T_EXDATA and T_GOEXDATA events cannot be returned.
7525		
7526	<i>t_open()</i>	No special considerations for NetBIOS transport providers, other than restrictions on the values returned in the <i>t_info</i> structure. These restrictions are described in <i>t_getinfo()</i> on page 63.
7527		
7528		
7529	<i>t_optmgmt()</i>	No special considerations for NetBIOS transport providers.
7530	<i>t_rcv()</i>	This function may only be used with connection-oriented transport endpoints. The <i>t_rcv()</i> function will fail if a user attempts a receive on a connectionless endpoint and <i>t_errno</i> will be set to [TNOTSUPPORT].
7531		
7532		
7533		The <i>flags</i> parameter will never be set to T_EXPEDITED, as expedited data is not supported.
7534		
7535		Data transfer in the NetBIOS environment is record-oriented, and the transport user should expect to see usage of the T_MORE flag when the message size exceeds the available buffer size.
7536		
7537		
7538	<i>t_rcvconnect()</i>	The NetBIOS name of the transport user responding to the previous connection request is provided in the <i>call</i> parameter (<i>call->addr.buf</i>), as described in Section D.5 on page 242.
7539		
7540		
7541		No user data may be returned to the caller (<i>call->udata.len=0</i> on return).
7542		This function may only be used with connection-oriented transport endpoints. The <i>t_rcvconnect()</i> function will fail if a user attempts to establish a connection on a connectionless endpoint and <i>t_errno</i> will be set to [TNOTSUPPORT].
7543		
7544		
7545	<i>t_rcvdis()</i>	The following disconnect reason codes are valid for any implementation of a NetBIOS provider under XTI:
7546		
7547		<pre>#define NB_ABORT 0x18 /* session ended abnormally */</pre>
7548		<pre>#define NB_CLOSED 0x0A /* session closed */</pre>
7549		<pre>#define NB_NOANSWER 0x14 /* no answer (cannot find */</pre>
7550		<pre>/* name called */</pre>
7551		<pre>#define NB_OPREJ 0x12 /* session open rejected */</pre>
7552		These definitions should be included in <xti.h> .
7553	<i>t_rcvrel()</i>	As described in Section D.6 on page 243, a T_ORDREL event will never occur in the T_DATAXFER state, but only in the T_OUTREL state. A transport user thus has only to prepare for a call to <i>t_rcvrel()</i> if it previously initiated a connection release by calling <i>t_sndrel()</i> . As a side effect, the state T_INREL is unreachable for the transport user.
7554		
7555		
7556		
7557		
7558		If T_COTS_ORD is not supported by the underlying NetBIOS transport provider, this function will fail with <i>t_errno</i> set to [TNOTSUPPORT].
7559		
7560	<i>t_rcvudata()</i>	The NetBIOS name of the sending transport user is provided in the <i>unitdata</i> parameter (<i>unitdata->addr.buf</i>), as described in Section D.5 on page 242.
7561		
7562		The <i>fd</i> associated with the <i>t_rcvudata()</i> function must refer to a connectionless transport endpoint. The function will fail if a user attempts to receive on a connection-oriented endpoint and <i>t_errno</i> will be set to [TNOTSUPPORT].
7563		
7564		[TBADF] may be returned in the case that the NetBIOS name associated with the <i>fd</i> referenced in the <i>t_rcvudata()</i> function is no longer in the CAE system name table, as may occur as a result of the NetBIOS name conflict resolution
7565		
7566		
7567		

7568		process (for example, TOP/NetBIOS NameConflictAdvise indication).
7569		To receive a broadcast datagram, the endpoint must be bound to the NetBIOS name NB_BCAST_NAME.
7570		
7571	<i>t_rcvuderr()</i>	If attempted on a connectionless transport endpoint, this function will fail with <i>t_errno</i> set to [TNOUDERR], as no NetBIOS unit data error codes are defined. If attempted on a connection-oriented transport endpoint, this function will fail with <i>t_errno</i> set to [TNOTSUPPORT].
7572		
7573		
7574		
7575	<i>t_snd()</i>	The T_EXPEDITED flag may not be set, as NetBIOS does not support expedited data transfer.
7576		
7577		This function may only be used with connection-oriented transport endpoints. The <i>t_snd()</i> function will fail if a user attempts a send on a connectionless endpoint and <i>t_errno</i> will be set to [TNOTSUPPORT].
7578		
7579		
7580		The maximum value of the <i>nbytes</i> parameter is determined by the maximum TSDU size allowed by the transport provider. The maximum TSDU size can be obtained from the <i>t_getinfo()</i> call.
7581		
7582		
7583		Data transfer in the NetBIOS environment is record-oriented. The transport user can use the <i>T_MORE</i> flag in order to fragment a TSDU and send it via multiple calls to <i>t_snd()</i> . See <i>t_snd()</i> on page 93 for more details.
7584		
7585		
7586		NetBIOS does not support the notion of expedited data. A call to <i>t_snd()</i> with the <i>T_EXPEDITED</i> flag will fail with <i>t_errno</i> set to [TBADDDATA].
7587		
7588		If the NetBIOS provider has received a HANGUP request from the remote user and still has receive data to deliver to the local user, XTI may not detect the HANGUP situation during a call to <i>t_snd()</i> . The actions that are taken are implementation-dependent:
7589		
7590		
7591		
7592		<ul style="list-style-type: none"> • <i>t_snd()</i> might fail with <i>t_errno</i> set to [TPROTO]
7593		
7594		<ul style="list-style-type: none"> • <i>t_snd()</i> might succeed, although the data is discarded by the transport provider, and an implementation-dependent error (generated by the NetBIOS provider) will result on a subsequent XTI call. This could be a [TSYSERR], a [TPROTO] or a connection release indication after all the receive data has been delivered.
7595		
7596		
7597		
7598	<i>t_snddis()</i>	The <i>t_snddis()</i> function initiates an abortive connection release. The function returns immediately. Outstanding send and receive data may be discarded. See Section D.6 on page 243 for further details.
7599		
7600		
7601		No user data may be sent in the disconnect request (call->udata.len=0).
7602		This function may only be used with connection-oriented transport endpoints. The <i>t_snddis()</i> function will fail if a user attempts a disconnect request on a connectionless endpoint and <i>t_errno</i> will be set to [TNOTSUPPORT].
7603		
7604		

7605	<i>t_sndrel()</i>	The <i>t_sndrel()</i> function initiates the NetBIOS release mechanism that attempts to complete outstanding sends within a timeout period before the connection is released. The function returns immediately. The transport user is informed by T_ORDREL when all sends have been completed and the connection has been closed successfully. If, however, the timeout occurs, the transport user is informed by T_DISIN and an appropriate disconnect reason code. See Section D.6 on page 243 for further details.
7606		
7607		
7608		
7609		
7610		
7611		
7612		If the NetBIOS transport provider did not return T_COTS_ORD with <i>t_open()</i> , this function will fail with <i>t_errno</i> set to [TNOTSUPPORT].
7613		
7614	<i>t_sndudata()</i>	The NetBIOS name of the destination transport user is provided in the <i>unitdata</i> parameter (<i>unitdata->addr.buf</i>), as described in Section D.5 on page 242.
7615		
7616		
7617		The <i>fd</i> associated with the <i>t_sndudata()</i> function must refer to a connectionless transport endpoint. The function will fail if a user attempts this function on a connection-oriented endpoint and <i>t_errno</i> will be set to [TNOTSUPPORT].
7618		[TBADF] may be returned in the case that the NetBIOS name associated with the <i>fd</i> referenced in the <i>t_sndudata()</i> function is no longer in the CAE system name table, as may occur as a result of the NetBIOS name conflict resolution process (for example, TOP/NetBIOS NameConflictAdvise indication).
7619		
7620		
7621		
7622		
7623		
7624		To send a broadcast datagram, the NetBIOS name in the NetBIOS address structure provided in <i>unitdata->addr.buf</i> must be NB_BCAST_NAME.
7625		
7626	<i>t_strerror()</i>	No special considerations for NetBIOS transport providers.
7627	<i>t_sync()</i>	No special considerations for NetBIOS transport providers.
7628	<i>t_unbind()</i>	No special considerations for NetBIOS transport providers.
7629		It is assumed that the NetBIOS transport provider will release the NetBIOS name associated with the endpoint if this is the only endpoint bound to this name.
7630		
7631		

XTI and TLI

7632

7633 XTI is based on the SVID Issue 2, Volume III, Networking Services Extensions (see Referenced
7634 Documents).

7635 XTI provides refinement of the Transport Level Interface (TLI) where such refinement is
7636 considered necessary. This refinement takes the form of:

- 7637 • additional commentary or explanatory text, in cases where the TLI text is either ambiguous
7638 or not sufficiently detailed
- 7639 • modifications to the interface, to cater for service and protocol problems which have been
7640 fully considered. In this case, it must be emphasised that such modifications are kept to an
7641 absolute minimum, and are intended to avoid any fundamental changes to the interface
7642 defined by TLI
- 7643 • the removal of dependencies on specific UNIX versions and specific transport providers.

7644 **E.1 Restrictions Concerning the Use of XTI**

7645 It is important to bear in mind the following points when considering the use of XTI:

- 7646 • It was stated that XTI “recommends” a subset of the total set of functions and facilities
7647 defined in TLI, and also that XTI introduces modifications to some of these functions and/or
7648 facilities where this is considered essential. For these reasons, an application which is written
7649 in conformance to XTI may not be immediately portable to work over a provider which has
7650 been written in conformance to TLI.
- 7651 • XTI does not address management aspects of the interface, that is:
 - 7652 — how addressing may be done in such a way that an application is truly portable
 - 7653 — no selection and/or negotiation of service and protocol characteristics.

7654 For addressing, the same is also true for TLI. In this case, it is envisaged that addresses will
7655 be managed by a higher-level directory function. For options selection and/or negotiation,
7656 XTI attempts to define a basic mechanism by which such information may be passed across
7657 the transport service interface, although again, this selection/negotiation may be done by a
7658 higher-level management function (rather than directly by the user). Since address structure
7659 is not currently defined, the user protocol address is system-dependent.

7660 E.2 Relationship between XTI and TLI

7661 The following features can be considered as XTI extensions to the System V Release 3 version of
7662 TLI:

- 7663 • Some functions may return more error types. The use of the [TOUTSTATE] error is
7664 generalised to almost all protocol functions.
- 7665 • The transport provider identifier has been generalised to remove the dependence on a device
7666 driver implementation.
- 7667 • Additional events have been defined to help applications make full use of the asynchronous
7668 features of the interface.
- 7669 • Additional features have been introduced to *t_snd()*, *t_sndrel()* and *t_rcvrel()* to allow fuller
7670 use of TCP transport providers.
- 7671 • Usage of options for certain types of transport service has been defined to increase
7672 application portability.
- 7673 • Because most XTI functions require read/write access to the transport provider, the usage of
7674 flags O_RDONLY and O_WRONLY has been withdrawn from the XTI.
- 7675 • XTI checks the value of *qlen* and prevents an application from waiting forever when issuing
7676 *t_listen()*.
- 7677 • XTI allows an application to call *t_accept()* with a *resfd* which is not bound to a local address.
- 7678 • XTI provides the additional utility functions *t_strerror()* and *t_getprotaddr()*.

Headers and Definitions for XTI

7679

7680 Section 7.1 on page 47 contains a normative requirement that the contents and structures found
7681 in this appendix appear in the <xti.h> header.

```

7682  /*
7683  * The following are the error codes needed by both the kernel
7684  * level transport providers and the user level library.
7685  */

7686  #define TBADADDR      1  /* incorrect addr format */
7687  #define TBADOPT      2  /* incorrect option format */
7688  #define TACCES       3  /* incorrect permissions */
7689  #define TBADF        4  /* illegal transport fd */
7690  #define TNOADDR      5  /* couldn't allocate addr */
7691  #define TOUTSTATE    6  /* out of state */
7692  #define TBADSEQ      7  /* bad call sequence number */
7693  #define TSYSEERR     8  /* system error */
7694  #define TLOOK        9  /* event requires attention */
7695  #define TBADDATA     10 /* illegal amount of data */
7696  #define TBUFOVFLW    11 /* buffer not large enough */
7697  #define TFLOW        12 /* flow control */
7698  #define TNODATA      13 /* no data */
7699  #define TNODIS       14 /* discon_ind not found on queue */
7700  #define TNOUDERR     15 /* unitdata error not found */
7701  #define TBADFLAG     16 /* bad flags */
7702  #define TNOREL       17 /* no ord rel found on queue */
7703  #define TNOTSUPPORT  18 /* primitive/action not supported */
7704  #define TSTATECHNG   19 /* state is in process of changing */
7705  #define TNOSTRUCTYPE 20 /* unsupported struct-type requested */
7706  #define TBADNAME     21 /* invalid transport provider name */
7707  #define TBADQLEN     22 /* qlen is zero */
7708  #define TADDRBUSY    23 /* address in use */
7709  #define TINDOUT      24 /* outstanding connection indications */
7710  #define TPROVMISMATCH 25 /* transport provider mismatch */
7711  #define TRESQLEN     26 /* resfd specified to accept w/qlen >0 */
7712  #define TRESADDR     27 /* resfd not bound to same addr as fd */
7713  #define TQFULL       28 /* incoming connection queue full */
7714  #define TPROTO       29 /* XTI protocol error */

```

```

7715  /*
7716  * The following are the events returned.
7717  */

7718  #define T_LISTEN      0x0001  /* connection indication received */
7719  #define T_CONNECT    0x0002  /* connect confirmation received */
7720  #define T_DATA       0x0004  /* normal data received */
7721  #define T_EXDATA     0x0008  /* expedited data received */
7722  #define T_DISCONNECT 0x0010  /* disconnect received */
7723  #define T_UDERR      0x0040  /* datagram error indication */
7724  #define T_ORDREL     0x0080  /* orderly release indication */
7725  #define T_GODATA     0x0100  /* sending normal data is again possible */
7726  #define T_GOEXDATA   0x0200  /* sending expedited data is again */
7727  /* possible */

7728  /*
7729  * The following are the flag definitions needed by the
7730  * user level library routines.
7731  */

7732  #define T_MORE       0x001  /* more data */
7733  #define T_EXPEDITED  0x002  /* expedited data */
7734  #define T_NEGOTIATE  0x004  /* set opts */
7735  #define T_CHECK      0x008  /* check opts */
7736  #define T_DEFAULT    0x010  /* get default opts */
7737  #define T_SUCCESS    0x020  /* successful */
7738  #define T_FAILURE    0x040  /* failure */
7739  #define T_CURRENT    0x080  /* get current options */
7740  #define T_PARTSUCCESS 0x100  /* partial success */
7741  #define T_READONLY   0x200  /* read-only */
7742  #define T_NOTSUPPORT 0x400  /* not supported */

7743  /*
7744  * XTI error return.
7745  */

7746  extern int t_errno;

7747  /* XTI LIBRARY FUNCTIONS */

7748  UX #ifndef _XOPEN_SOURCE_EXTENDED
7749  /* XTI Library Function: t_accept - accept a connect request*/
7750  extern int t_accept();
7751  /* XTI Library Function: t_alloc - allocate a library structure*/
7752  extern char *t_alloc();
7753  /* XTI Library Function: t_bind - bind an address to a transport endpoint*/
7754  extern int t_bind();
7755  /* XTI Library Function: t_close - close a transport endpoint*/
7756  extern int t_close();
7757  /* XTI Library Function: t_connect - establish a connection */
7758  extern int t_connect();
7759  /* XTI Library Function: t_error - produce error message*/
7760  extern int t_error();
7761  /* XTI Library Function: t_free - free a library structure*/
7762  extern int t_free();
7763  /* XTI Library Function: t_getprotaddr - get protocol addresses*/
7764  extern int t_getprotaddr();
7765  /* XTI Library Function: t_getinfo - get protocol-specific service */
7766  /* information*/
7767  extern int t_getinfo();

```

```

7768     /* XTI Library Function: t_getstate - get the current state*/
7769     extern int t_getstate();
7770     /* XTI Library Function: t_listen - listen for a connect indication*/
7771     extern int t_listen();
7772     /* XTI Library Function: t_look - look at current event on a transport */
7773                                     /* endpoint*/
7774     extern int t_look();
7775     /* XTI Library Function: t_open - establish a transport endpoint*/
7776     extern int t_open();
7777     /* XTI Library Function: t_optmgmt - manage options for a transport */
7778                                     /* endpoint*/
7779     extern int t_optmgmt();
7780     /* XTI Library Function: t_rcv - receive data or expedited data on a */
7781                                     /* connection*/
7782     extern int t_rcv();
7783     /* XTI Library Function: t_rcvdis - retrieve information from disconnect*/
7784     extern int t_rcvdis();
7785     /* XTI Library Function: t_rcvrel - acknowledge receipt of */
7786     /* an orderly release indication */
7787     extern int t_rcvrel();
7788     /* XTI Library Function: t_rcvudata - receive a data unit*/
7789     extern int t_rcvudata();
7790     /* XTI Library Function: t_rcvuderr - receive a unit data error indication*/
7791     extern int t_rcvuderr();
7792     /* XTI Library Function: t_snd - send data or expedited data over a */
7793                                     /* connection */
7794     extern int t_snd();
7795     /* XTI Library Function: t_snddis - send user-initiated disconnect request*/
7796     extern int t_snddis();
7797     /* XTI Library Function: t_sndrel - initiate an orderly release*/
7798     extern int t_sndrel();
7799     /* XTI Library Function: t_sndudata - send a data unit*/
7800     extern int t_sndudata();
7801     /* XTI Library Function: t_strerror - generate error message string */
7802     extern char *t_strerror();
7803     /* XTI Library Function: t_sync - synchronise transport library*/
7804     extern int t_sync();
7805     /* XTI Library Function: t_unbind - disable a transport endpoint*/
7806     extern int t_unbind();
7807 ux  #else
7808     extern int t_accept(int, int, struct t_call *);
7809     extern char *t_alloc(int, int, int);
7810     extern int t_bind(int, struct t_bind *, struct t_bind *);
7811     extern int t_close(int);
7812     extern int t_connect(int, struct t_call *, struct t_call *);
7813     extern int t_error(char *);
7814     extern int t_free(char *, int);
7815     extern int t_getinfo(int, struct t_info *);
7816     extern int t_getprotaddr(int, struct t_bind *, struct t_bind *);
7817     extern int t_getstate(int);
7818     extern int t_listen(int, struct t_call *);
7819     extern int t_look(int);
7820     extern int t_open(char *, int, struct t_info *);
7821     extern int t_optmgmt(int, struct t_optmgmt *, struct t_optmgmt *);
7822     extern int t_rcv(int, char *, unsigned int, int *);
7823     extern int t_rcvconnect(int, struct t_call *);
7824     extern int t_rcvdis(int, struct t_discon *);
7825     extern int t_rcvrel(int);
7826     extern int t_rcvudata(int, struct t_unitdata *, int *);

```

```

7827 extern int t_rcvuderr(int, struct t_uderr *);
7828 extern char *t_strerror(int);
7829 extern int t_snd(int, char *, unsigned int , int);
7830 extern int t_snddis(int, struct t_call *);
7831 extern int t_sndrel(int);
7832 extern int t_sndudata(int, struct t_unitdata *);
7833 extern int t_sync(int);
7834 extern int t_unbind(int);
7835 #endif

7836 /*
7837  * Protocol-specific service limits.
7838  */
7839 struct t_info {
7840     long  addr;      /* max size of the transport protocol address */
7841     long  options;   /* max number of bytes of protocol-specific options */
7842     long  tsdu;      /* max size of a transport service data unit */
7843     long  etsdu;     /* max size of expedited transport service data unit */
7844     long  connect;   /* max amount of data allowed on connection */
7845                 /* establishment functions */
7846     long  discon;    /* max data allowed on t_snddis and t_rcvdis functions */
7847     long  servtype; /* service type supported by transport provider */
7848     long  flags;     /* other info about the transport provider */
7849 };

7850 /*
7851  * Service type defines.
7852  */

7853 #define T_COTS      01 /* connection-oriented transport service */
7854 #define T_COTS_ORD  02 /* connection-oriented with orderly release */
7855 #define T_CLTS     03 /* connectionless transport service */

7856 /*
7857  * Flags defines (other info about the transport provider).
7858  */

7859 #define T_SENDFLAGS 0x001 /* supports 0-length TSDUs */

7860 /*
7861  * netbuf structure.
7862  */

7863 struct netbuf {
7864     unsigned int  maxlen;
7865     unsigned int  len;
7866     char          *buf;
7867 };

```


Headers and Definitions for XTI

```
7868      /*
7869      * t_opthdr structure
7870      */
7871      struct t_opthdr {
7872          unsigned long len;          /* total length of option; that is,      */
7873                                     /* sizeof (struct t_opthdr) + length */
7874                                     /* of option value in bytes          */
7875          unsigned long level;        /* protocol affected                  */
7876          unsigned long name;         /* option name                        */
7877          unsigned long status;       /* status value                       */
7878      /* followed by the option value */
7879      };

7880      /*
7881      * t_bind - format of the address and options arguments of bind.
7882      */

7883      struct t_bind {
7884          struct netbuf  addr;
7885          unsigned      qlen;
7886      };

7887      /*
7888      * Options management structure.
7889      */

7890      struct t_optmgmt {
7891          struct netbuf  opt;
7892          long           flags;
7893      };

7894      /*
7895      * Disconnect structure.
7896      */

7897      struct t_discon {
7898          struct netbuf  udata;        /* user data */
7899          int            reason;       /* reason code */
7900          int            sequence;     /* sequence number */
7901      };

7902      /*
7903      * Call structure.
7904      */

7905      struct t_call {
7906          struct netbuf  addr;         /* address */
7907          struct netbuf  opt;         /* options */
7908          struct netbuf  udata;       /* user data */
7909          int            sequence;     /* sequence number */
7910      };
```

```

7911      /*
7912      * Datagram structure.
7913      */

7914      struct t_unitdata {
7915          struct netbuf  addr;    /* address */
7916          struct netbuf  opt;    /* options */
7917          struct netbuf  udata;  /* user data */
7918      };

7919      /*
7920      * Unitdata error structure.
7921      */

7922      struct t_uderr {
7923          struct netbuf  addr;    /* address */
7924          struct netbuf  opt;    /* options */
7925          long          error;   /* error code */
7926      };

7927      /*
7928      * The following are structure types used when dynamically
7929      * allocating the above structures via t_alloc().
7930      */

7931      #define T_BIND          1    /* struct t_bind */
7932      #define T_OPTMGMT      2    /* struct t_optmgmt */
7933      #define T_CALL        3    /* struct t_call */
7934      #define T_DIS         4    /* struct t_discon */
7935      #define T_UNITDATA    5    /* struct t_unitdata */
7936      #define T_UDERROR     6    /* struct t_uderr */
7937      #define T_INFO        7    /* struct t_info */

7938      /*
7939      * The following bits specify which fields of the above
7940      * structures should be allocated by t_alloc().
7941      */

7942      #define T_ADDR         0x01  /* address */
7943      #define T_OPT          0x02  /* options */
7944      #define T_UDATA       0x04  /* user data */
7945      #define T_ALL         0xffff /* all the above fields supported */

```

Headers and Definitions for XTI

```
7946      /*
7947      * The following are the states for the user.
7948      */

7949      #define T_UNBND      1 /* unbound */
7950      #define T_IDLE      2 /* idle */
7951      #define T_OUTCON    3 /* outgoing connection pending */
7952      #define T_INCON     4 /* incoming connection pending */
7953      #define T_DATAXFER  5 /* data transfer */
7954      #define T_OUTREL    6 /* outgoing release pending */
7955      #define T_INREL     7 /* incoming release pending */

7956      /*
7957      * General purpose defines.
7958      */

7959      #define T_YES        1
7960      #define T_NO         0
7961      #define T_UNUSED     -1
7962      #define T_NULL       0
7963      #define T_ABSREQ     0x8000
7964      #define T_INFINITE   -1
7965      #define T_INVALID    -2

7966      /* T_INFINITE and T_INVALID are values of t_info */
7967      /*
7968      * General definitions for option management
7969      */
7970      #define T_UNSPEC     (~0 - 2) /* applicable to u_long, long, char .. */
7971      #define T_ALLOPT     0
7972      #define T_ALIGN(p)  (((unsigned long)(p) + (sizeof (long) - 1)) \
7973                          & ~(sizeof (long) - 1))
7974      #define OPT_NEXTHDR( pbuf, buflen, popt) \
7975          (((char *) (popt) + T_ALIGN( (popt)->len ) < \
7976           (char *) (pbuf) + (buflen)) ? \
7977           (struct t_opthdr *) ((char *) (popt) + T_ALIGN( (popt)->len )) : \
7978           (struct t_opthdr *) 0 )

7979          /* OPTIONS ON XTI LEVEL */
7980      /* XTI-level */

7981      #define XTI_GENERIC  0xffff

7982      /*
7983      * XTI-level Options
7984      */

7985      #define XTI_DEBUG    0x0001 /* enable debugging */
7986      #define XTI_LINGER   0x0080 /* linger on close if data present */
7987      #define XTI_RCVBUF   0x1002 /* receive buffer size */
7988      #define XTI_RCVLOWAT 0x1004 /* receive low-water mark */
7989      #define XTI_SNDBUF   0x1001 /* send buffer size */
```

```

7990     #define   XTI_SNDLOWAT   0x1003   /* send low-water mark */

7991     /*
7992     * Structure used with linger option.
7993     */
7994     struct t_linger {
7995         long   l_onoff;       /* option on/off */
7996         long   l_linger;     /* linger time */
7997     };

7998         /* SPECIFIC ISO OPTION AND MANAGEMENT PARAMETERS */

7999     /*
8000     * Definition of the ISO transport classes
8001     */

8002     #define   T_CLASS0   0
8003     #define   T_CLASS1   1
8004     #define   T_CLASS2   2
8005     #define   T_CLASS3   3
8006     #define   T_CLASS4   4

8007     /*
8008     * Definition of the priorities.
8009     */

8010     #define   T_PRITOP   0
8011     #define   T_PRIHIGH  1
8012     #define   T_PRIMID  2
8013     #define   T_PRILOW  3
8014     #define   T_PRIDFLT  4

8015     /*
8016     * Definitions of the protection levels
8017     */

8018     #define   T_NOPROTECT      1
8019     #define   T_PASSIVEPROTECT 2
8020     #define   T_ACTIVEPROTECT 4

8021     /*
8022     * Default value for the length of TPDU's.
8023     */

8024     #define   T_LTPDUDFLT  128   /* define obsolete in XPG4 */

8025     /*
8026     * rate structure.
8027     */
8028     struct rate {
8029         long targetvalue;       /* target value */
8030         long minacceptvalue;   /* value of minimum acceptable quality */
8031     };

```

Headers and Definitions for XTI

```
8032     /*
8033     * reqvalue structure.
8034     */
8035     struct reqvalue {
8036         struct rate    called;    /* called rate */
8037         struct rate    calling;   /* calling rate */
8038     };
8039     /*
8040     * thrpt structure.
8041     */
8042     struct thrpt {
8043         struct reqvalue    maxthrpt;    /* maximum throughput */
8044         struct reqvalue    avgthrpt;    /* average throughput */
8045     };
8046     /*
8047     * transdel structure
8048     */
8049     struct transdel {
8050         struct reqvalue    maxdel;    /* maximum transit delay */
8051         struct reqvalue    avgdel;    /* average transit delay */
8052     };
8053     /*
8054     * Protocol Levels
8055     */
8056     #define    ISO_TP    0x0100
8057     /*
8058     * Options for Quality of Service and Expedited Data (ISO 8072:1986)
8059     */
8060     #define    TCO_THROUGHPUT                0x0001
8061     #define    TCO_TRANSDEL                  0x0002
8062     #define    TCO_RESERRORRATE              0x0003
8063     #define    TCO_TRANSFFAILPROB           0x0004
8064     #define    TCO_ESTFAILPROB              0x0005
8065     #define    TCO_RELFAILPROB              0x0006
8066     #define    TCO_ESTDELAY                  0x0007
8067     #define    TCO_RELDELAY                  0x0008
8068     #define    TCO_CONNRRESIL                0x0009
8069     #define    TCO_PROTECTION                0x000a
8070     #define    TCO_PRIORITY                  0x000b
8071     #define    TCO_EXPD                      0x000c
8072     #define    TCL_TRANSDEL                  0x000d
8073     #define    TCL_RESERRORRATE              TCO_RESERRORRATE
8074     #define    TCL_PROTECTION                TCO_PROTECTION
8075     #define    TCL_PRIORITY                  TCO_PRIORITY
```

```

8076      /*
8077      * Management Options
8078      */

8079      #define   TCO_LTPDU           0x0100
8080      #define   TCO_ACKTIME        0x0200
8081      #define   TCO_REASTIME       0x0300
8082      #define   TCO_EXTFORM        0x0400
8083      #define   TCO_FLOWCTRL       0x0500
8084      #define   TCO_CHECKSUM       0x0600
8085      #define   TCO_NETEXP         0x0700
8086      #define   TCO_NETRECPTCF     0x0800
8087      #define   TCO_PREFCLASS      0x0900
8088      #define   TCO_ALTCLASS1      0x0a00
8089      #define   TCO_ALTCLASS2      0x0b00
8090      #define   TCO_ALTCLASS3      0x0c00
8091      #define   TCO_ALTCLASS4      0x0d00

8092      #define   TCL_CHECKSUM       TCO_CHECKSUM

8093      /* INTERNET SPECIFIC ENVIRONMENT */

8094      /*
8095      * TCP level
8096      */

8097      #define   INET_TCP           0x6

8098      /*
8099      *TCP-level Options
8100      */

8101      #define   TCP_NODELAY         0x1 /* don't delay packets to coalesce */
8102      #define   TCP_MAXSEG         0x2 /* get maximum segment size */
8103      #define   TCP_KEEPAALIVE     0x8 /* check, if connections are alive */

8104      /*
8105      * Structure used with TCP_KEEPAALIVE option.
8106      */
8107      struct t_kpalive {
8108          long    kp_onoff;        /* option on/off */
8109          long    kp_timeout;      /* timeout in minutes */
8110      };

8111      #define   T_GARBAGE           0x02

8112      /*
8113      * UDP level
8114      */

8115      #define   INET_UDP           0x11

8116      /*
8117      * UDP-level Options
8118      */

```

Headers and Definitions for XTI

```
8119     #define    UDP_CHECKSUM    TCO_CHECKSUM    /* checksum computation */
8120
8121     /*
8122     * IP level
8123     */
8124
8125     #define    INET_IP    0x0
8126
8127     /*
8128     * IP-level Options
8129     */
8130
8131     #define    IP_OPTIONS        0x1    /* IP per-packet options */
8132     #define    IP_TOS            0x2    /* IP per-packet type of service */
8133     #define    IP_TTL            0x3    /* IP per-packet time to live /
8134     #define    IP_REUSEADDR      0x4    /* allow local address reuse */
8135     #define    IP_DONTROUTE      0x10   /* just use interface addresses */
8136     #define    IP_BROADCAST      0x20   /* permit sending of broadcast msgs */
8137
8138     /*
8139     * IP_TOS precedence levels
8140     */
8141
8142     #define    T_ROUTINE         0
8143     #define    T_PRIORITY        1
8144     #define    T_IMMEDIATE       2
8145     #define    T_FLASH           3
8146     #define    T_OVERRIDEFLASH   4
8147     #define    T_CRITIC_ECP      5
8148     #define    T_INETCONTROL     6
8149     #define    T_NETCONTROL      7
8150
8151     /*
8152     * IP_TOS type of service
8153     */
8154
8155     #define    T_NOTOS           0
8156     #define    T_LDELAY          1 << 4
8157     #define    T_HITHRPT         1 << 3
8158     #define    T_HIRES           1 << 2
8159
8160     #define    SET_TOS(prec, tos) ((0x7 & (prec)) << 5 | (0x1c & (tos)))
```


Abbreviations

8152

8153	CO	Connection-oriented
8154	CL	Connectionless
8155	EM	Event Management
8156	ETSDU	Expedited Transport Service Data Unit
8157	ISO	International Organization for Standardization
8158	OSI	Open System Interconnection
8159	SVID	System V Interface Definition
8160	TC	Transport Connection
8161	TCP	Transmission Control Protocol
8162	TLI	Transport Level Interface
8163	TSAP	Transport Service Access Point
8164	TSDU	Transport Service Data Unit
8165	UDP	User Datagram Protocol
8166	XTI	X/Open Transport Interface
8167	XEM	X/Open Event Management Interface

Minimum OSI Functionality (Preliminary Specification)

8168

8169 H.1 General

8170 The purpose of this specification is to provide a simple API exposing a minimum set of OSI
8171 Upper Layers functionality (mOSI).

8172 H.1.1 Rationale for using XTI-mOSI

8173 This appendix uses the concept of a minimal set of OSI upper layer facilities that support basic
8174 communication applications. A Basic Communication Application simply requires the ability to
8175 open and close communications with a peer and to send and receive messages with a peer.

8176 XTI-mOSI is designed specifically for Basic Communication Applications that are in one of these
8177 categories:

- 8178 • applications that are to be migrated from the Internet world (TCP or UDP) or from a
8179 NetBIOS environment to OSI
- 8180 • applications accessing the OSI transport service that wish to migrate to an OSI seven-layer,
8181 conformant environment
- 8182 • applications that require a simple octet-stream connection between peer processes. The
8183 benefit of XTI-mOSI to these applications is that it extends the family of *transport services* that
8184 are available via a single, protocol independent, API.

8185 H.1.2 Migrant Applications

8186 For the first kind of applications (those migrating to OSI or intended to work over a variety of
8187 *transport* mechanisms), the migration effort will be greatly simplified if they were already using
8188 XTI — mOSI offers several new options, but, as described later in this section, default values are
8189 generally provided.

8190 In addition to applications already using XTI, the X Window System (X) and Internet Protocol
8191 Suite applications (in general) are examples of potential Migrant applications.

8192 H.1.3 OSI Functionality

8193 mOSI is suited to applications that require only the Minimal Upper Layer facilities which are
8194 described in the profile (ISO/IEC pDISP 11188 — Common Upper Layer Requirements, Part 3:
8195 Minimal OSI upper layer facilities — OIW/EWOS working documents). These are:

- 8196 • ACSE Kernel functional unit
- 8197 • Presentation Kernel functional unit
- 8198 • Session Kernel and Full Duplex functional units.

8199 The XTI-mOSI interface provides access to OSI ACSE and Presentation services. With mOSI, the
8200 optional parameters available to the application have been selected with the intent of facilitating
8201 interoperability and diagnostic of problems. They are described later in this section.

8202 Most applications only need the Kernel functionality. This is even true for most of the OSI
8203 standard applications: Remote Database Access (RDA), Directory (X.500), FTAM without

8204 recovery, OSI Distributed Transaction Processing (TP) without 2-phase commitment, OSI
8205 Management.

8206 **H.1.4 mOSI API versus XAP**

8207 X/Open has developed XAP (ACSE/Presentation API), which offers full access to ACSE and
8208 Presentation functionality and is well suited for system programmers/integrators needing to use
8209 all of its functionality (including minor and major synchronisation points...).

8210 XAP needs to be used when some of the following pieces of functionality (not available with
8211 mOSI) are required:

- 8212 • use of Functional Units different from Kernel and Full Duplex
- 8213 • access to AP and AE invocation identifiers, to session connection identifier (which may be
8214 useful with some of the resynchronisation/activity management functional units).

8215 In general, XAP will be used for applications targetted at the OSI environment that may need to
8216 take advantage of additional OSI facilities in the future. XAP is a flexible, extensible API;
8217 extensions to cover OSI Remote Operation Services (ROSE) and OSI Transaction Processing are
8218 under development.

8219 **H.1.5 Upper Layers Functionality Exposed via mOSI**

8220 These are presented as they are exposed via mOSI options and specific parameters.

8221 *H.1.5.1 Naming and Addressing Information used by mOSI*

8222 The addr structure (used in t_bind, t_connect, t_accept) is a combined naming and addressing
8223 datatype, identifying one end or the other of the association.

8224 The address part is a Presentation Address. The Calling and Called addresses are required
8225 parameters while the use of a Responding address is optional.

8226 The name part (Application Process (AP) Title and Application Entity (AE) Qualifier) is always
8227 optional.

8228 ISO Directory facilities, when available, can relate the name parts (identifying specific
8229 applications) to the addresses of the real locations where they can be accessed.

8230 The general format of the addr structure can be found in Section H.5 on page 283, while its
8231 precise structure is implementation dependent.

8232 *H.1.5.2 XTI Options Specific to mOSI*

- 8233 • Application Context Name

8234 An application context name identifies a set of tasks to be performed by an application. It is
8235 exchanged during association establishment with the purpose of conveying a common
8236 understanding of the work to be done.

8237 This parameter is exposed to offer some negotiation capabilities to the application and to
8238 increase the chances of interoperability.

8239 When receiving a non suitable or unknown value from a peer application, the application
8240 may propose an alternate value or decide to terminate prematurely the association.

8241 A default value (in the form of an Object Identifier) is provided, identifying a generic XTI-
8242 mOSI application. Its value can be found in Section H.5 on page 283.

- 8243 • Presentation Contexts
- 8244 A presentation context is the association of an abstract syntax with a transfer syntax. The
8245 presentation context is used by the application to identify how the data is structured and by
8246 the OSI Application Layer to identify how the data should be encoded/decoded.
- 8247 A *generic* presentation context is defined for a stream-oriented, unstructured, data transfer
8248 service with *null* encoding:
- 8249 *abstract syntax*: The single data type of this abstract syntax is a sequence of octets that are
8250 defined in the application protocol specification as being consecutive octets on a stream-
8251 oriented transport mechanism, without regard for any semantic or other boundaries.
- 8252 *transfer syntax*: The data value shall be represented as an octet-aligned presentation data
8253 value. If two or more data values are concatenated together they are considered to be a
8254 single (longer) data value. (This is the *null* encoding rule).
- 8255 The value of the Object Identifiers for this *generic* presentation context can be found in
8256 Section H.5 on page 283.
- 8257 • Presentation Context Definition List, Result List, Defined Context Set
- 8258 As negotiation occurs between the peer OSI Application layers, the presentation context(s)
8259 proposed by the application may not be accepted.
- 8260 The Presentation Context Definition Result List indicates, for each of the proposed
8261 presentation context, if it is accepted or, if not, provides a reason code; the application may
8262 choose to terminate the association prematurely if it does not suit its requirements.

8263 **H.2 Options**

8264 Options are formatted according to the structure `t_opthdr` as described in Chapter 6 on page 35.
 8265 An OSI provider compliant to this specification supports all, none or a subset of the options
 8266 defined in Section H.2.1. An implementation may restrict the use of any of the options by
 8267 offering them in privileged or `read_only` mode.

8268 An explanation of when an application may benefit from using the XTI options specific to mOSI
 8269 can be found in Section H.1 on page 267.

8270 **H.2.1 ACSE/Presentation Connection-oriented Service**

8271 The protocol level for all subsequent options is `ISO_APCO`.

8272 All options are association-related (see Chapter 6 on page 35. They may be negotiated in the XTI
 8273 states `T_IDLE` and `T_INCON`, and are read-only in all other states except `T_UNINIT`. The
 8274 structures referenced are specified in Section H.5 on page 283.

Option Name	Type of Option Value	Legal Option Value	Meaning
AP_CNTX_NAME	Object identifier item (see Section H.5 on page 283)	see text default: see text	Application Context Name
AP_PCDL	Presentation Context Definition list (see Section H.5 on page 283)	see text default: see text	Presentation Context Definition List
AP_PCDRL	Presentation Context Definition Result list (see Section H.5 on page 283)	see text default: none	Presentation Context Definition Result List
AP_MCPC	unsigned long	<code>T_YES/T_NO</code> default: <code>T_NO</code>	multiple choice presentation contexts

8291 **Table H-1** APCO-level Options

8292 **Further Remarks**

- 8293 • Application Context Name

8294 A default value (for a *generic* XTI-mOSI application) is provided. It is defined in Section H.5
 8295 on page 283.

8296 The application may choose to propose, through this option, a value different from the
 8297 default one. The application may also use this option to check the value returned by the peer
 8298 application and decide if the association should be kept or terminated.

- 8299 • Presentation Context Definition List

8300 A default is provided: a list with one presentation context (the stream oriented, unstructured,
 8301 data transfer service with *null* encoding — this is described in section Section H.1 on page
 8302 267. The abstract syntax is the default abstract syntax and the transfer syntax is the default
 8303 transfer syntax, as specified in Section H.5 on page 283.

- 8304 • Presentation Context Definition Result List
- 8305 The codes for the result of negotiation and reason for rejection are defined in Section H.5 on
- 8306 page 283. The responding application, after reading this option, may choose to continue or
- 8307 terminate the association.
- 8308 • Multiple Choice Presentation Contexts
- 8309 The default behaviour (AP_MCPC set to T_NO) frees the application from having to make
- 8310 choices for encoding of user-data parameters. In that case, the responder is requested to pick
- 8311 up one of the user-data presentation contexts offered by the initiator; this rule is enforced by
- 8312 the API (Note that the ACSE presentation context is required, but this is handled by the API
- 8313 implementation.
- 8314 If a unique user presentation context is too limiting, the application may prefer to perform all
- 8315 encodings (all PDV list, with indication of the PCI used). In this case, on the association
- 8316 responder side, AP_MCPC must be set to T_YES to let the API pass all the user-data
- 8317 presentation contexts offered to the application, responsible for the negotiation (otherwise
- 8318 the API will select a unique one).
- 8319 On the association initiator or responder side, when AP_MCPC is set to T_YES, the first user
- 8320 data buffer of each more bit sequence of data buffers starts with a long datatype containing
- 8321 an identifier corresponding to the PCI.

8322 **Management Options**

8323 No management options are defined.

8324 **H.2.2 ACSE/Presentation Connectionless Service**

8325 The protocol level for all subsequent options is ISO_APCL.

8326 All options are association-related (see Chapter 6 on page 35). They may be negotiated in all XTI

8327 states except T_UNINIT. The structures referenced are specified in Section H.5 on page 283.

Option Name	Type of Option Value	Legal Option Value	Meaning
AP_CNTX_NAME	Object identifier item (see Section H.5 on page 283)	see text default: see text	Application Context Name
AP_PCDL	Presentation Context Definition list (see Section H.5 on page 283)	see text default: see text	Presentation Context Definition List

8338 **Table H-2 APCL-level Options**

- 8339 **Further Remarks**
- 8340 • Application Context Name
- 8341 A default value (for a *generic* XTI-mOSI application) is provided. It is defined in Section H.5
8342 on page 283.
- 8343 The application may choose to propose, through this option, a value different from the
8344 default one. The application may also use this option to check the value returned by the peer
8345 application and decide if the datagram should be kept or discarded.
- 8346 • Presentation Context Definition List
- 8347 In connectionless mode, the transfer syntaxes are not negotiated. Their use are determined
8348 by the sending application entity, and must be acceptable by the receiving application entity.
8349 A default value is provided by XTI: a list with one element, the *generic* presentation context
8350 (the stream-oriented, unstructured, data transfer service with *null* encoding described in
8351 Section H.1 on page 267). The corresponding abstract and transfer syntaxes are specified in
8352 Section H.5 on page 283.
- 8353 **Management Options**
- 8354 No management options are defined.
- 8355 **H.2.3 Transport Service Options**
- 8356 Some of the options defined for XTI ISO Transport Connection-oriented Service or Transport
8357 Connectionless Service may be made available to mOSI users: the Options for Quality of
8358 Service.
- 8359 These Options are defined in Section A.2.1.1 on page 190 and Section A.2.2.1 on page 194. The
8360 Quality of Service parameters are passed directly by the OSI Upper Layers to the Transport
8361 Layer. These options can thus be used to specify OSI Upper Layers quality of service parameters
8362 via XTI.
- 8363 This facility is implementation dependent. An attempt to specify an unsupported option will
8364 return with the status field set to T_NOTSUPPORT.
- 8365 None of these options are available with an ISO-over-TCP transport provider.

8366 **H.3 Functions**

8367	<i>t_accept()</i>	If <i>fd</i> is not equal to <i>resfd</i> , <i>resfd</i> should either be in state T_UNBND or be in state T_IDLE and be bound to the same address as <i>fd</i> with the <i>qlen</i> parameter set to 0.
8368		
8369		
8370		The <i>addr</i> parameter passed to/returned from <i>t_bind</i> when <i>resfd</i> is bound may be different from the <i>addr</i> parameter corresponding to <i>fd</i> .
8371		
8372		The <i>opt</i> parameter may be used to change the Application Context Name received.
8373		
8374	<i>t_alloc()</i>	No special considerations for mOSI providers.
8375	<i>t_bind()</i>	The <i>addr</i> field of the <i>t_bind()</i> structure represents the local presentation address and optionally the local AP Title and AE Qualifier (see Section H.1 on page 267 and Section H.5 on page 283 for more details).
8376		
8377		
8378		This local <i>addr</i> field is used, depending on the XTI primitive, as the calling, called or responding address, the called address being different from the responding address only when two different file descriptors (<i>fd</i> , <i>resfd</i>), bound to different addresses, are used.
8379		
8380		
8381		
8382	<i>t_close()</i>	Any connections that are still active at the endpoint are abnormally terminated. The peer applications will be informed of the disconnection by a [T_DISCONNECT] event. The value of the disconnect reason will be [AC_ABRT_PEER].
8383		
8384		
8385		
8386	<i>t_connect()</i>	The <i>sndcall->addr</i> structure specifies the Called Presentation Address. The <i>rcvcall->addr</i> structure specifies the Responding Presentation Address. The structure may also be used to assign values for the Called AP Title and Called AE Qualifier.
8387		
8388		
8389		
8390		Before the call, the <i>sndcall->opt</i> structure may be used to request an Application Context name or Presentation Context different from the default value.
8391		
8392		
8393	<i>t_error()</i>	No special considerations for mOSI providers.
8394	<i>t_free()</i>	No special considerations for mOSI providers.
8395	<i>t_getinfo()</i>	The information supported by <i>t_getinfo()</i> reflects the characteristics of the <i>transport</i> connection, or if no connection is established, the default characteristics of the underlying OSI layers. In all possible states except T_DATAXFER, the function <i>t_getinfo()</i> returns in the parameter <i>info</i> the same information as was returned by <i>t_open()</i> . In state T_DATAXFER, however, the information returned in <i>info->connect</i> and <i>info->discon</i> may differ.
8396		
8397		
8398		
8399		
8400		
8401		The parameters of the <i>t_getinfo()</i> function are summarised in the table below.

8402
8403
8404
8405
8406
8407
8408
8409
8410
8411
8412
8413

Parameters	Before call	After call	
		Connection-oriented	Connectionless
fd	x	/	/
info->addr	/	x	x
info->options	/	x	x
info->tsdu	/	-1	-1
info->etsdu	/	-2	-2
info->connect	/	x	-2
info->discon	/	x	-2
info->servtype	/	T_COTS_ORD	T_CLTS
info->flags	/	0	0

8414

x equals an integral number greater than 0.

8415
8416

The values of the parameters in the `t_info` structure for the `t_getinfo()` function reflect the mOSI provider particularities.

8417

- connect, discon

8418
8419
8420
8421
8422

The values returned in `info->connect` and `info->discon` in state `T_DATAXFER` may differ from the values returned by `t_open()`: negotiation takes place during association establishment and, as a result, these values may be reduced. For `info->connect`, this change of value may be indicated by the provider, but is of little use to the application.

8423

- flags

8424
8425

mOSI does not support sending of TSDU of zero length, so this value equals 0.

8426
8427

`t_getprotaddr()` The protocol addresses are naming and addressing parameters as defined in Section H.1 on page 267 and Section H.5 on page 283.

8428

`t_getstate()` No special considerations for mOSI providers.

8429
8430

`t_listen()` The `call->addr` structure contains the remote Calling Presentation Address, and optionally the remote Calling AP Title and AE Qualifier.

8431
8432

`t_look()` Since expedited data is not supported for a mOSI provider, `T_EXDATA` and `T_GOEXDATA` events cannot occur.

8433
8434
8435

`t_open()` `t_open()` is called as the first step in the initialisation of a *transport* endpoint. This function returns various default characteristics of the underlying OSI layers.

8436

The parameters of the `t_open()` function are summarised in the table below.

	Parameters	Before call	After call	
			Connection-oriented	Connectionless
8437				
8438				
8439				
8440	name	x	/	/
8441	oflag	/	/	/
8442	info->addr	/	x	x
8443	info->options	/	x	x
8444	info->tsdu	/	-1	-1
8445	info->etsdu	/	-2	-2
8446	info->connect	/	x	-2
8447	info->discon	/	x	-2
8448	info->servtype	/	T_COTS_ORD	T_CLTS
8449	info->flags	/	0	0

8450 x equals an integral number greater than 0.

8451 The values of the parameters in the *t_info* structure reflect mOSI limitations as
8452 follows:

- 8453 • connect, discon

8454 These values are limited by the version of the session supported by the
8455 mOSI provider, and are generally much larger than those supported by an
8456 ISO Transport or TCP provider.

- 8457 • flags

8458 mOSI does not support sending of *TSDU* of zero length, so this value
8459 equals 0.

8460 **Note:** The name (device file) parameter passed to *t_open()* will differ when
8461 the application accesses an mOSI provider or an ISO Transport
8462 provider.

8463 *t_optmgt()* The options available with mOSI providers are described in section Section
8464 H.2 on page 270.

8465 *t_rcv()* The flags parameter will never be set to [T_EXPEDITED], as expedited data
8466 transfer is not supported.

8467 *t_rcvconnect()* The call->addr structure specifies the remote Responding Presentation
8468 Address.

8469 The call->opt structure may also contain an Application Context Name
8470 and/or Presentation Context Definition Result List.

8471 *t_rcvdis()* Possible values for disconnect reason codes are specified in Section H.5 on
8472 page 283.

8473 *t_rcvrel()* With this primitive, user data cannot be received on normal release: any user
8474 data in the received flow is discarded (see Section H.6 on page 287, XTI
8475 Change Request 20-01).

8476 *t_rcvudata()* The unitdata->addr structure specifies the remote Presentation address, and
8477 optionally the remote AP Title and AE Qualifier. If the T_MORE flag is set, an
8478 additional *t_rcvudata()* call is needed to retrieve the entire A-UNIT-DATA
8479 service unit. Only normal data is returned via the *t_rcvudata()* call.

8480 *t_rcvuderr()* This function is not supported by a mOSI provider since badly formed A-
8481 UNIT-DATA APDUs are discarded.

8482	<i>t_snd()</i>	Zero-length TSDUs are not supported.
8483		Since expedited data transfer is not supported for a mOSI provider, the
8484		parameter flags shall not have [T_EXPEDITED] set.
8485	<i>t_snddis()</i>	No special considerations for mOSI providers.
8486	<i>t_sndrel()</i>	With this primitive, user data cannot be sent on normal release (see Section
8487		H.6 on page 287, XTI Change Request 20-01).
8488	<i>t_sndudata()</i>	The <i>unitdata->addr</i> structure specifies the remote Presentation address, and
8489		optionally the remote AP Title and AE Qualifier. Only normal data is sent via
8490		the <i>t_sndudata()</i> call.
8491	<i>t_strerror()</i>	No special considerations for mOSI providers.
8492	<i>t_sync()</i>	No special considerations for mOSI providers.
8493	<i>t_unbind()</i>	No special considerations for mOSI providers.

8494 **H.4 Implementors' Notes**8495 **H.4.1 Upper Layers FUs, Versions and Protocol Mechanisms**

8496 The implementation negotiates:

8497 Session: Kernel, Full Duplex, version 2, or version 1 if version 2 not supported, no
8498 segmentation.8499 Other session protocol mechanisms are out of scope, except Basic
8500 Concatenation which is mandatory and transparent to the application.

8501 Presentation: Kernel, Normal Mode

8502 ACSE: Kernel

8503 If invalid (non-negotiable) options are requested by the peer and detected by the provider once
8504 the association is already established (such as the ACSE presentation context missing in the
8505 Defined Context Set), the association is rejected via an A-(P)-ABORT generated by the
8506 implementation.8507 **H.4.2 Mandatory and Optional Parameters**8508 • If the Local Presentation Address is not passed to *t_bind()* in req->addr, then it is returned in
8509 ret->addr.

8510 • The following parameters must be explicitly set by the application:

8511 — Remote (called) Presentation Address (in *t_connect()*, sndcall->addr).8512 — If, in *t_accept()*, a new accepting endpoint is specified (*resfd* != *fd*), a Presentation Address
8513 must be bound to the new accepting endpoint (the Responding Presentation Address). If
8514 the same endpoint is used, the Responding Presentation Address is equal to the Local
8515 (Called) Presentation Address.8516 • The following parameters are mandatory for the protocol machine, but default values are
8517 provided. If the application does not wish to set the corresponding parameter, the default
8518 value will be used. The default value may be changed through *t_optmgt* (see Section H.2 on
8519 page 270):

8520 — Application Context Name (opt parameter)

8521 — Presentation Contexts (opt parameter).

8522 The presentation context of ACSE is required and used. The user should not request it as
8523 the implementation will insert it automatically in the context list.8524 If the user does not specifically request an Application Context name via the opt
8525 parameter of *t_accept* (that is, for the A-Associate response), the implementation uses the
8526 Application Context name that was received in the A-Associate indication.8527 • The following parameters are optional for the protocol and default values of null are defined.
8528 If the application does not set them otherwise, they are omitted from the outgoing protocol
8529 stream.8530 — local AP-title (in *t_bind()*, req->addr)8531 — called AP-title (in *t_connect()*, sndcall->addr)8532 — responding AP-title (if *t_accept()* specifies a new accepting endpoint *resfd*, in the protocol
8533 address bound to *resfd*)

- 8534 — local AE-qualifier (in *t_bind()*, req->addr)
- 8535 — called AE-qualifier (in *t_connect()*, sndcall->addr)
- 8536 — responding AE-qualifier (if *t_accept()* specifies a new accepting endpoint *resfd*, in the
- 8537 protocol address bound to *resfd*).
- 8538 • The following parameters are optional for the protocol machine and not supported through
- 8539 the XTI interface. Their handling is implementation-defined. Received values in the
- 8540 incoming protocol stream, if any, are discarded:
- 8541 — ACSE Protocol Version (default= version 1)
- 8542 — Presentation Protocol Version (default= version 1)
- 8543 — ACSE Implementation Information
- 8544 — AP invocation identifiers (called, calling, and responding)
- 8545 — AE invocation identifiers (called, calling, and responding)
- 8546 — Session connection identifiers.

8547 H.4.3 Mapping XTI Functions to ACSE/Presentation Services

8548 In the following tables, for a given primitive, the presence of each parameter in the protocol flow
 8549 is described in the OSI column by M or O, as specified in Annex A of *Common Upper Layer*
 8550 *Requirements, Part 3: Minimal OSI upper layer facilities - OIW/EWOS working document*.
 8551 Connectionless protocols are not yet included in CURL part 3.

8552 For items sent, the status column is from the *Sender Status for Category II specification* column in
 8553 Annex A of CURL - part 3. The Receiver Status is always set to *M*, as parameters which are
 8554 optional on the sending side must be acceptable (that is, not generate aborts) on the receiving
 8555 side, even if they are subsequently to be ignored.

8556 M Mandatory: Support for the feature is mandatory — as sender, as receiver or as both sender
 8557 and receiver.

8558 O Optional: Support for the item is the option of the referencing specification — as sender, as
 8559 receiver or as both sender and receiver.

8560 H.4.3.1 Connection-oriented Services

8561 Association Establishment (successful, unsuccessful)

8562 **Note:** XTI does not support the concept of a negative association establishment; that is, the
 8563 equivalent of a negative A-ASSOCIATE response. That is, an XTI-mOSI
 8564 implementation does not generate an AARE- APDU.

8565 To reject an association request, the responding application issues *t_snddis()*, which is mapped to
 8566 a A-ABORT.

8567 However, a negative A-ASSOCIATE confirm (AARE- APDU) may be received from a non-XTI
 8568 OSI peer. The negative A-ASSOCIATE confirm event is mapped to *t_rcvdis()*.

8569

Table H-3 Association Establishment

8570

8571

8572

8573

8574

8575

8576

8577

8578

8579

8580

8581

8582

8583

8584

8585

8586

8587

8588

8589

8590

8591

8592

8593

8594

8595

8596

8597

8598

8599

8600

8601

8602

8603

8604

8605

8606

8607

8608

8609

8610

8611

8612

8613

8614

8615

8616

8617

8618

8619

XTI call	Parameter	Service	Parameter	OSI
t_connect		A-ASSOCIATE req		
	sndcall->addr		Called Presentation Address	M
	sndcall->addr (1)		Called AP Title	O
	sndcall->addr (1)		Called AE Qualifier	O
	sndcall->opt (2)		Application Context Name	M
	sndcall->opt (3)		P-context Definition List	M
	sndcall->udata		User Information	O
{t_bind}	req ret->addr		Calling Presentation Address	M
{t_bind}	req ret->addr		Calling AP Title	O
{t_bind}	req ret->addr		Calling AE Qualifier	O
t_listen		A-ASSOCIATE ind		
	call->addr		Calling Presentation Address	M
	call->addr (1)		Calling AP Title	M
	call->addr (1)		Calling AE Qualifier	M
	call->opt		Application Context Name	M
	call->opt (4)		P-context Definition List	M
	call->udata		User Information	M
{t_bind}	req ret->addr		Called Presentation Address	M
{t_bind}	req ret->addr (1)		Called AP Title	M
{t_bind}	req ret->addr (1)		Called AE Qualifier	M
t_accept		A-ASSOCIATE rsp+		
	call->addr		not used: Calling Presentation Address	O
	call->opt		Application Context Name	M
	call->opt		P-context Definition Result List	M
	call->udata		User Information	O
{internal}	::="accepted"		Result	M
{t_bind}	req ret->addr		Responding Presentation Address	M
{t_bind}	req ret->addr (1)		Responding AP Title	O
{t_bind}	req ret->addr (1)		Responding AE Qualifier	M
not sent		A-ASSOCIATE rsp-		
t_connect	(synchronous mode)	A-ASSOCIATE cnf+		
	rcvcall->addr		Responding Presentation Address	M
	rcvcall->addr		Responding AP Title	M
	rcvcall->addr		Responding AE Qualifier	M
	rcvcall->opt		Application Context Name	M
	rcvcall->opt		P-context Definition Result List	M
	rcvcall->udata		User Information	M
{internal}	::="accepted"		Result	M
{internal}	::="ACSE service-user"		Result Source	M
t_rcvconnect	(asynchronous mode)	A-ASSOCIATE cnf+		
	call->addr		Responding Presentation Address	M
	call->addr		Responding AP Title	M
	call->addr		Responding AE Qualifier	M
	call->opt		Application Context Name	M
	call->opt		P-context Definition Result List	M
	call->udata		User Information	M
{discarded}	::="accepted"		Result	M
{discarded}	::="ACSE service-user"		Result Source-diagnostic	M
t_rcvdis		A-ASSOCIATE cnf-		

	XTI call	Parameter	Service	Parameter	OSI
8620					
8621		discon->udata		User Information	M
8622		discon->reason (5)		Result	M
8623	{internal}	ACSE serv-user pres serv-prov		Result Source-diagnostic	M
8624		{discarded}		Application Context Name	M
8625		{discarded}		P-context Definition Result List	M

8626 **Notes:**

- 8627 (1) if either the AP title or AE qualifier is selected for sending, the other must be
- 8628 selected.
- 8629 (2) *sndcall*→*opt* or, if no option specified, default value
- 8630 (3) *sndcall*→*opt* or, if no option specified, default value, with ACSE added by
- 8631 provider
- 8632 (4) *call*→*opt* with ACSE context removed from the list passed to user
- 8633 (5) combines Result and Result Source-diagnostic

8634 **Data Transfer**

	XTI call	Parameter	Service	Parameter	OSI
8635					
8636					
8637	t_snd		P-DATA req		
8638		buf		User Data	M
8639	t_rcv		P-DATA ind		
8640		buf		User Data	M

8641 **Table H-4** Data Transfer

8642 **Association Release (orderly, abortive)**

8643 This table makes the assumption that the XTI-mOSI provider supports the orderly release
 8644 facility with user data (*t_sndrel2()* and *t_rcvrel2()*, see Section H.6 on page 287). When this is not
 8645 the case, User Information is not sent, Reason is supplied via an internal mechanism with A-
 8646 RELEASE request and response, User Information and Reason received in A-RELEASE
 8647 indication and confirmation are discarded.

	XTI call	Parameter	Service	Parameter	OSI
8648	t_sndrel2		A-RELEASE req		
8649		reldata->reason		Reason	M
8650		reldata->udata		User Information	O
8651	t_rcvrel2		A-RELEASE ind		
8652		reldata->reason		Reason	M
8653		reldata->udata		User Information	M
8654	t_sndrel2		A-RELEASE rsp		
8655		reldata->reason		Reason	M
8656		reldata->udata		User Information	O
8657	t_rcvrel2		A-RELEASE cnf		
8658		reldata->reason		Reason	M
8659		reldata->udata		User Information	M
8660	t_snddis		A-ABORT req		
8661		n/s		Diagnostic	M
8662		call->udata		User Information	O
8663	t_rcvdis		A-ABORT ind		
8664		discon->reason		Diagnostic	M
8665		discon->udata		User Information	M
8666	t_rcvdis		A-P-ABORT ind		
8667		discon->reason		Diagnostic	M

8670

Table H-5 Association Release

8671 H.4.3.2 Connectionless Services

8672
86738674
8675
8676
8677
8678
8679
8680
8681
8682

XTI call	Parameter	Service	Parameter	OSI
t_sndudata		A-UNIT-DATA source		
	unitdata->addr		Called Presentation Address	M
	unitdata->addr		Called AP Title	O
	unitdata->addr		Called AE Qualifier	O
	unitdata->opt (1)		Application Context Name	M
	unitdata->opt (2)		P-context Definition List	O (4)
	unitdata->udata		User Information	M
{t_bind}	req ret->addr		Calling Presentation Address	M
{t_bind}	req ret->addr		Calling AP Title	O
{t_bind}	req ret->addr		Calling AE Qualifier	O
t_rcvudata		A-UNIT-DATA sink		
	unitdata->addr		Calling Presentation Address	M
	unitdata->addr		Calling AP Title	M
	unitdata->addr		Calling AE Qualifier	M
	unitdata->opt		Application Context Name	M
	unitdata->opt (3)		P-context Definition List	M (4)
	unitdata->udata		User Information	M
{t_bind}	req ret->addr		Called Presentation Address	M
{t_bind}	req ret->addr		Called AP Title	M
{t_bind}	req ret->addr		Called AE Qualifier	M

8693

Table H-6 Connectionless-mode ACSE Service

8694

Notes:8695
8696
8697
8698
8699
8700

- (1) unitdata->opt or, if no option specified, default value
- (2) unitdata->opt or, if no option specified, default value, with ACSE added by provider
- (3) unitdata->opt with ACSE context removed from the list passed to user
- (4) ISO 8822 AM1 (connectionless Presentation service) defines this parameter as *user option*; CURL part 3 does not currently cover connection-less services.

8701 **H.5 Complements to <xti.h>**8702 **SPECIFIC ISO ACSE/PRESENTATION OPTIONS**8703 **Naming and Addressing Datatype**

8704 The buf[] part of the addr structure is an mosiaddr structure defined in the following way:

```

8705     struct t_mosiaddr {
8706         unsigned int    osi_apt_len;
8707         unsigned int    osi_aeq_len;
8708         unsigned int    osi_paddr_len;
8709         unsigned char   osi_addr[ MAX_ADDR ];
8710     }

```

8711 Where:

8712 the *apt* address starts at osi_addr[0]8713 the *aeq* address starts at osi_addr[T_ALIGN(osi_apt_len)]8714 the *paddr* is at osi_addr[T_ALIGN(osi_apt_len) + T_ALIGN(osi_aeq_len)]

8715 MAX_ADDR is an implementation-defined constant.

8716 The application is responsible for encoding/decoding the AP title and AE qualifier; alternatively,
8717 a lookup routine may be provided (outside the scope of this specification).8718 **ACSE/Presentation Option Levels and Names**

```

8719 #define ISO_APCO          0x0200
8720 #define ISO_APCL         0x0300
8721 #define AP_CNTX_NAME     0x1
8722 #define AP_PCDL          0x2
8723 #define AP_PCDRL         0x3
8724 #define AP_MCPC          0x4

```

8725 **Object Identifier Representation within Options**8726 The presentation context definition list and application context both utilise object identifiers. An
8727 object identifier is held as a variable length item of the following form:

```

8728     | len | object_value... | // |
8729     -----
8730     ulong                                ^
8731                                         |
8732                                         alignment
8733                                         characters

```

8734 The application is responsible for encoding/decoding the Object id value; alternatively, a lookup
8735 routine may be provided (outside the scope of this specification).

8736 **Application Context Name Option**

8737 The application context name option consists of an object identifier item as defined above.

8738 **Presentation Context Definition List Option**8739 The presentation context definition list option is used to propose one or more presentation
8740 contexts, giving their abstract syntax and allowable transfer syntaxes.8741 The presentation context definition list option is a variable size option consisting of a long giving
8742 the number of presentation contexts followed that number of presentation context definition
8743 elements.8744 Each presentation context definition element consists of a presentation context item header
8745 defined as:

```
8746     struct t_ap_pcd_hdr {
8747         long pci;
8748         long t_sytx_size;
8749     }
```

8750 followed by an object identifier item for the abstract syntax and *t_sytx_size* number of object
8751 identifier items, one for each of the proposed transfer syntaxes.8752 **Presentation Context Definition Result List Option**8753 A presentation definition context result list option gives the result of negotiation, and consists of
8754 a long giving the number of presentation contexts followed by that number of presentation
8755 context definition result elements, each defined as:

```
8756     struct t_ap_pcdr {
8757         long res;           /* result of negotiation */
8758         long prov_rsn;     /* reason for rejection */
8759     }

8760     /*
8761     * codes for res and prov_rsn
8762     */

8763     #define PCDRL_ACCPT      0x0
8764                             /*pres. context accepted          */
8765     #define PCDRL_USER_REJ  0x1
8766                             /*pres. context rejected by peer application */
8767     #define PCDRL_PREJ_RSN_NSPEC 0x0100
8768                             /*prov. reject: no reason specified          */
8769     #define PCDRL_PREJ_A_SYTX_NSUP 0x0101
8770                             /*prov. reject: abstract syntax not supported*/
8771     #define PCDRL_PREJ_T_SYTX_NSUP 0x0102
8772                             /*prov. reject:transfer syntax not supported */
8773     #define PCDRL_PREJ_LMT_DCS_EXCEED 0x0103
8774                             /*prov. reject: local limit on DCS exceeded */
```

8775 For the default abstract syntax, transfer syntax and application context, this Appendix uses
8776 object identifiers which are specified in the profile (ISO/IEC pDISP 11188 - Common Upper
8777 Layer Requirements, Part 3: Minimal OSI upper layer facilities - OIW/EWOS working
8778 document). Thus the descriptions provided in this Appendix are informative only.

8779 Default Abstract Syntax for mOSI

8780 The following OBJECT IDENTIFIER have been defined in CURL part 3:

8781 {iso(1) standard(0) curl(11188) mosi(3) default-abstract-syntax(1) version(1)}

8782 This object identifier can be used as the abstract syntax when the application protocol (above
8783 ACSE) can be treated as single presentation data values (PDVs). Each PDV is a sequence of
8784 consecutive octets without regard for semantic or other boundaries. The object identifier may
8785 also be used when, for pragmatic reasons, the actual abstract syntax of the application is not
8786 identified in Presentation Layer negotiation.

8787 Notes:

- 8788 1. Applications specified using ASN.1 should not use the default abstract syntax.
- 8789 2. As this object identifier is used by all applications using the default abstract
8790 syntax for mOSI, it cannot be used to differentiate between applications. One of
8791 the ACSE parameters; for example, AE Title or Presentation address, may be used
8792 to differentiate between applications.

8793 Default Transfer Syntax for mOSI

8794 If the default transfer syntax and the abstract syntax are identical, the OBJECT IDENTIFIER for
8795 the default abstract syntax is used. If they are not identical, the OBJECT identifier for the default
8796 transfer syntax is:

8797 {iso(1) standard(0) curl(11188) mosi(3) default-transfer-syntax(2) version(1)}

8798 **Note:** In the presentation data value of the PDV list of Presentation Protocol or in the
8799 encoding of User Information of ACSE Protocol, only *octet-aligned* or *arbitrary* can be
8800 used for default transfer syntax for mOSI. *Single-ASN1-type* cannot be used for default
8801 transfer syntax for mOSI.

8802 Default Application Context for mOSI

8803 The following OBJECT IDENTIFIER has been defined in CURL part 3:

8804 {iso(1) standard(0) curl(11188) mosi(3) default-application-context(3) version(1)}

8805 This application context supports the execution of any application using the default abstract
8806 syntax for mOSI.

8807 **Reason Codes for Disconnections**

```
8808           #define AC_U_AARE_NONE    0x0001     /*connection rejected by     */
8809                                        /*peer user: no reason given */
8810           #define AC_U_AARE_ACN     0x0002     /*connection rejected:     */
8811                                        /*application context name  */
8812                                        /*not supported             */
8813           #define AC_U_AARE_APT     0x0003     /*connection rejected:     */
8814                                        /*AP title not recognised   */
8815           #define AC_U_AARE_AEQ     0x0005     /*connection rejected:     */
8816                                        /*AE qualifier not recognised*/
8817           #define AC_U_AARE_PEER_AUTH 0x000e    /*connection rejected:     */
8818                                        /*authentication required   */
8819           #define AC_P_ABRT_NSPEC   0x0011     /*aborted by peer provider: */
8820                                        /*no reason given           */
8821           #define AC_P_AARE_VERSION  0x0012     /*connection rejected:     */
8822                                        /*no common version         */
```

8823 Other reason codes may be specified as implementation defined constants. In order to be
8824 portable, an application should not interpret such information, which should only be used for
8825 troubleshooting purposes.

8826 **H.6 XTI mOSI CR**

8827 Several references are made in this Appendix to XTI CR 20-01. This change request proposes
8828 extending the functionality of the X/Open Transport interface. It is intended that this change
8829 proposal will be decided along with other proposals to extend XTI functionality, in the near
8830 future.

8831 The content of XTI CR 20-01 is presented in this section, for convenience. If accepted, it will be
8832 removed from this Appendix when the extended functionality it proposes is incorporated with
8833 other XTI functions.

8834 Document: X/Open Transport Interface (XTI), CAE Specification, Version 2

8835 Change Number: 20-01

8836 Title: Orderly release with user data

8837 Qualifier: Major Technical

8838 Rationale: XTI permits to send and receive user data with the
8839 abortive release primitives (t_snddis, t_rcvdis) but not
8840 with the orderly release primitives (t_sndrel, t_rcvrel).

8841 This is consistent with TCP specifications.

8842 For ISO ACSE, providing an orderly release mechanism, user
8843 data is a parameter of the release service. OSI
8844 applications that use A-RELEASE user data are FTAM and VT
8845 (Virtual Terminal); for ROSE applications, the argument
8846 of UNBIND is mapped to A-RELEASE user data.

8847 When mapping XTI primitives to ACSE/Presentation
8848 (XTI-mOSI Appendix), disconnect user data may thus be
8849 received from peer applications. Three alternatives are
8850 possible:

- 8851 1. Discard user data - detrimental to those applications
8852 that want to receive user data from the peer, but
8853 still possible for the others.
 - 8854 2. User data delivered (via t_rcv); this would happen
8855 just before the T_ORDREL event, with introduction of a
8856 new flag T_ORDREL_DATA.
 - 8857 3. Addition of a new primitive, t_rcvrel2, with user data
8858 parameter - this method is more straightforward than
8859 alternative 2 (both for the application and the
8860 library implementation), and additive (does not break
8861 existing applications).
- 8862 See part 1 of the proposed change.

8863 If user data can be received with t_rcvrel2, it makes sense
8864 to propose a similar handling for the emission of user data,
8865 thus a new primitive t_sndrel2, with user data parameter, is
8866 proposed below (if this new primitive is not present, the
8867 only alternative is that user data is not supported by the
8868 provider). See part 2 of the proposed change.

8869 Support of these new primitives needs to be indicated,
8870 see part 3 of the change.

8871 Change:

8872 Part 1: Add a new manual page after t_rcvrel():
8873 -----

8874 Name:
8875 t_rcvrel2 - receipt of an orderly release indication or
8876 confirmation containing user data.

8877 SYNOPSIS
8878 #include <xti.h>

8879 int t_rcvrel2(fd, discon)
8880 int fd;
8881 struct t_discon *discon;

8882 DESCRIPTION

8883	Parameters	Before call		After call
8884	-----	-----	-----	-----
8885	fd	x		/
8886	discon->udata.maxlen	x		/
8887	discon->udata.len	/		x
8888	discon->udata.buf	?		(?)
8889	discon->reason	/		x
8890	discon->sequence	/		/

8891 This function is used to acknowledge receipt of an
8892 orderly release indication or confirmation and to
8893 retrieve any user data sent with the release. The
8894 argument fd identifies the local transport endpoint
8895 where the connection exists, and discon points to a
8896 t_discon structure containing the following members:

8897 struct netbuf udata;
8898 int reason;
8899 int sequence;

8900 After receipt of this indication, the user may not
8901 attempt to receive more data because such an attempt
8902 will block forever. However, the user may continue to
8903 send data over the connection if t_sndrel() or t_sndrel2()
8904 has not been called by the user.

8905 The field reason specifies the reason for the
8906 disconnect through a protocol-dependent reason code
8907 and udata identifies any user data that was sent with the
8908 disconnect; the field sequence is not used.

8909 If a user does not care if there is incoming data and
8910 does not need to know the value of reason, discon may
8911 be a null pointer, and any user data associated with
8912 the disconnect will be discarded.

8913 This function is an optional service of the transport
8914 provider, only supported by providers of service type
8915 T_COTS_ORD. The flag T_ORDRELDATA in the info->flag
8916 field returned by t_open or t_getinfo indicates that
8917 the provider does not discard received disconnect user data.

8918 This function may not be available on all systems.

8919 VALID STATES

8920 T_DATAXFER, T_OUTREL

8921 ERRORS

8922 On failure, t_errno is set to one of the following:

8923 [TBADF] The specified file descriptor does not refer
8924 to a transport endpoint.

8925 [TNOREL] No orderly release indication currently
8926 exists on the specified transport endpoint.

8927 [TLOOK] An asynchronous event has occurred on this
8928 transport endpoint and requires immediate
8929 attention.

8930 [TNOTSUPPORT] Orderly release is not supported by the
8931 underlying transport provider.

8932 [TSYSERR] A system error has occurred during execution
8933 of this function.

8934 [TOUTSTATE] The function was issued in the wrong
8935 sequence on the transport endpoint
8936 referenced by fd.

8937 [TPROTO] This error indicates that a communication
8938 problem has been detected between XTI and
8939 the transport provider for which there is
8940 no other suitable XTI(t_errno).

8941 [TBUFOVFLW] The number of bytes allocated for
8942 incoming data (maxlen) is greater than
8943 0 but not sufficient to store the data,
8944 and the disconnect information to be
8945 returned in discon will be discarded.
8946 The provider state, as seen by the
8947 user, will be changed as if the data
8948 was successfully retrieved.

8949 RETURN VALUE

8950 Upon successful completion, a value of 0 is returned.
8951 Otherwise, a value of -1 is returned and t_errno is set
8952 to indicate an error.

8953 SEE ALSO

8954 t_getinfo(), t_open(), t_sndrel2(), t_rcvrel(), t_sndrel().

8955 Part 2: Add a new manual page after t_sndrel():
 8956 -----

8957 Name:
 8958 t_sndrel2 - initiate/respond to an orderly release with
 8959 user data

8960 SYNOPSIS
 8961 #include <xti.h>

8962 int t_sndrel2(fd, discon)
 8963 int fd;
 8964 struct t_discon *discon;

8965 DESCRIPTION

8966 Parameters	8967 Before call	8968 After call
8969 fd	8970 x	8971 /
8972 discon->udata.maxlen	8973 /	8974 /
8975 discon->udata.len	8976 x	8977 /
8978 discon->udata.buf	8979 ?(?)	8980 /
8981 discon->reason	8982 ?	8983 /
8984 discon->sequence	8985 /	8986 /

8974 This function is used to initiate an orderly release or
 8975 to respond to an orderly release indication and to send
 8976 user data with the release. The argument fd identifies
 8977 the local transport endpoint where the connection
 8978 exists, and discon points to a t_discon structure
 8979 containing the following members:

```

8980     struct netbuf udata;
8981     int reason;
8982     int sequence;
  
```

8983 After calling t_sndrel2(), the user may not send any
 8984 more data over the connection. However, a user may
 8985 continue to receive data if an orderly release
 8986 indication has not been received.

8987 The field reason specifies the reason for the
 8988 disconnect through a protocol-dependent reason code
 8989 and udata identifies any user data that is sent with the
 8990 disconnect; the field sequence is not used.

8991 The udata structure specifies the user data to be sent
 8992 to the remote user. The amount of user data must not
 8993 exceed the limits supported by the transport provider,
 8994 as returned in the dicson filed of the info argument of
 8995 t_open() or t_getinfo(). If the len field of udata is
 8996 zero, no data will be sent to the remote user.

8997 If a user does not wish to send data and reason code to
 8998 the remote user, the value of discon may be a null
 8999 pointer.

9000 This function is an optional service of the transport
 9001 provider, only supported by providers of service type

9002 T_COTS_ORD. The flag T_ORDRELDATA in the info->flag
 9003 field returned by t_open or t_getinfo indicates that
 9004 the provider will accept to send disconnect user data.
 9005 This function may not be available on all systems.

9006 VALID STATES

9007 T_DATAXFER, T_INREL

9008 ERRORS

9009 On failure, t_errno is set to one of the following:

9010 [TBADF] The specified file descriptor does not refer
 9011 to a transport endpoint.

9012 [TFLOW] O_NONBLOCK was set, but the flow
 9013 control mechanism prevented the
 9014 transport provider from accepting the
 9015 function at this time.

9016 [TLOOK] An asynchronous event has occurred on this
 9017 transport endpoint and requires immediate
 9018 attention.

9019 [TNOTSUPPORT] Orderly release is not supported by the
 9020 underlying transport provider.

9021 [TOUTSTATE] The function was issued in the wrong
 9022 sequence on the transport endpoint
 9023 referenced by fd.

9024 [TSYSERR] A system error has occurred during execution
 9025 of this function.

9026 [TPROTO] This error indicates that a
 9027 communication problem has been detected
 9028 between XTI and the transport provider
 9029 for which there is no other suitable
 9030 XTI(t_errno).

9031 [TBADDATA] The amount of user data specified was
 9032 not within the bounds allowed by the
 9033 transport provider or the provider did not
 9034 return T_ORDRELDATA in the t_open flags.

9035 RETURN VALUE

9036 Upon successful completion, a value of 0 is returned.
 9037 Otherwise, a value of -1 is returned and t_errno is set
 9038 to indicate an error.

9039 SEE ALSO

9040 t_getinfo(), t_open(), t_rcvrel2(), t_rcvrel(), t_sndrel().

9041 Part 3: indication by provider of support of the new
9042 primitives
9043 -----

9044 - Section 4.3, XTI features:
9045 Change:
9046 "The orderly release mechanism (using t_sndrel() and
9047 t_rcvrel()) is supported only for T_COTS_ORD type providers."

9048 into:
9049 "The orderly release mechanism (using t_sndrel(), t_sndrel2(),
9050 t_rcvrel() and t_rcvrel2()) is supported only for T_COTS_ORD
9051 type providers."

9052 Other sections with editorial changes resulting from this CR:
9053 -----

9054 Mention t_rcvrel2(), t_sndrel2() in addition to existing
9055 functions in:
9056 - Table 3-1,
9057 - Section 4.1.4, Overview of Connection Release,
9058 - Section 4.3.1, XTI Functions versus Protocols,
9059 - Table 5-2,
9060 - Section 5.6, Events and TLOOK error indication.

SNA Transport Provider

9061

9062 **I.1 Introduction**

9063 This Appendix includes:

9064 • Protocol-specific information that is relevant for Systems Network Architecture (SNA)
9065 transport providers.

9066 It assumes native SNA users, that is, those prepared to use SNA addresses and other SNA
9067 transport characteristics (for example, mode name for specifying quality of service).

9068 • Information on the mapping of XTI functions to Full Duplex (FDX) LU 6.2.

9069 Systems that do not support LU 6.2 full duplex can simulate them using twin-opposed half-
9070 duplex conversations. Protocols for doing so will be published separately.

9071 The half-duplex verbs have been published for several years. The full-duplex verbs will be
9072 published in 1993. Copies are available⁹ on request.

9073 _____

9074 9. Until the full-duplex verbs are published in the public domain, copies of the relevant specification *CPI-C Full Duplex*
9075 *Conversations and Expedited Data, Nov 30 1992* may be requested from IBM Corporation, via X/Open.

9076 I.2 SNA Transport Protocol Information

9077 This section describes the protocol-specific information that is relevant for Systems Network
9078 Architecture (SNA) transport providers.

9079 I.2.1 General

9080 1. Protocol address

9081 For information about SNA addresses, see Section I.2.2 on page 295.

9082 2. Connection establishment

9083 Native SNA has no confirmed allocation protocol for full duplex conversations. When a
9084 conversation is allocated, the connection message is buffered and sent with the first data
9085 that is sent on the conversation. When the *t_connect()* or *t_rcvconnect()* function completes,
9086 connectivity has been established to the partner node, but not to the partner program.
9087 Since notification that the partner is not available may occur later, the disconnect reasons
9088 returned on *t_rcvdis()* include [SNA_CONNECTION_SETUP_FAILURE], indicating that
9089 the connection establishment never completed successfully.

9090 An SNA program that needs to know that the partner is up and running before it proceeds
9091 sending data must have its own user-level protocol to determine if this is so.

9092 3. Parallel connections

9093 LU 6.2 allows multiple, simultaneous connections between the same pair of addresses.
9094 The number of connections possible between two systems depends on limits defined by
9095 system administrators.

9096 4. Sending data of zero octets is supported.

9097 5. Expedited data

9098 In connection-oriented mode, expedited data transfer can be negotiated by the two
9099 transport providers during connection establishment. Expedited data transfer is
9100 supported if both transport providers support it. However negotiation between transport
9101 users is not supported. Therefore the expedited option is read-only.

9102 6. Orderly release

9103 The orderly release functions, *t_sndrel()* and *t_rcvrel()*, can be used for the orderly release
9104 facility of SNA, just as they are for TCP.

9105 7. SNA buffers data from multiple *t_snd()* functions until the SNA send buffer is full,
9106 allowing multiple records to be sent in one transmission. However, users sometimes have
9107 reasons for ensuring that a record is sent immediately. By setting the T_PUSH flag on the
9108 *t_snd()* function, the transport user causes data to be transferred without waiting for the
9109 buffer to be filled.

9110 In order to take advantage of the performance improvement that SNA buffering offers, the
9111 XTI user must set the SNA_ALWAYS_PUSH option to T_NO (default is T_YES). If this
9112 option is not set to T_NO, a push will be done for every *t_snd()* and the T_PUSH flag will
9113 have no effect.

9114 8. Programs migrated to SNA from other transport providers may want every *t_snd()* to
9115 cause a message to be sent immediately in order to match behaviour on the original
9116 provider. The default of this option is T_YES; thus the default is that a *t_snd()* will always
9117 be sent out immediately.

9118 **I.2.2 SNA Addresses**

9119 In an SNA environment, the protocol address always includes a network-ID-qualified logical
9120 unit (LU) name. This is the address of the node where the program resides.

9121 For the `t_connect()` and `t_sndudata()` functions, the address also contains a transaction program
9122 name (TPN), identifying the program addressed in the partner node. A file descriptor used to
9123 accept incoming connection requests should have a complete SNA name, including TPN, bound
9124 to it with `t_bind()`.

9125 A file descriptor used for outgoing connection requests may optionally have only a network-id-
9126 qualified LU name bound to it.

9127 Since the `t_listen()` returns only the LU name part of the address, this address is not adequate for
9128 opening up a connection back to the source. The transport user must know the TPN of its
9129 partner by some mechanism other than XTI services.

9130 However, `t_rcvudata()` returns the complete address of the partner that can be used to send a
9131 datagram back to it.

9132 An SNA address has the following structure. When the TPN is not included, the TPN length
9133 (`sna_tpn_length`) is set to zero, and the string that follows is null.

```
9134     /* The definitions for maximum LU name and netid lengths have specific */
9135     /* values because these maxima are a fixed SNA characteristic,      */
9136     /* not an implementation option. Maximum TP length is a implementation */
9137     /* option, although the maximum maximum is 64.                      */
```

```
9138     #define SNA_MAX_NETID_LEN      8
9139     #define SNA_MAX_LU_LEN        8
9140     #define SNA_MAX_TPN_LEN
```

```
9141     struct sna_addr
9142     {
9143         u_char  sna_netid      (SNA_MAX_NETID_LEN),
9144         u_char  sna_lu        (SNA_MAX_LU_LEN),
9145         u_short sna_tpn_len,   /* less than or equal to SNA_MAX_TPN_LEN */
9146         u_char  sna_tpn      (sna_tpn_len)
9147     }
```

9148 **Notes:**

- 9149 1. network-identifier (`sna_netid`): The address can contain either an SNA network
9150 identifier or the defined value, `SYS_NET`, which indicates that the predefined
9151 network identifier associated with the local system should be used.
- 9152 2. IBM Corporation provides a registration facility for SNA network identifiers to
9153 guarantee global uniqueness. (See IBM document G325-6025-0, SNA Network
9154 Registry).
- 9155 3. LU name (`sna_lu`): The address can contain either a specific LU name or the
9156 defined value, `SYS_LU`, which indicates that the system default LU name is to be
9157 used.
- 9158 4. LU name and network identifier fields are fixed length. For values shorter than 8
9159 characters, they are blank filled to the right.
- 9160 5. Transaction program name (`sna_tpn`): This field can take one of three values:
 - 9161 — **Null value:** No transaction program name is to be associated with the file
9162 descriptor.

- 9163 This is adequate for file descriptors used for outgoing connection requests.
- 9164 If no transaction program is associated with a file descriptor when a *t_listen()*,
9165 *t_rcvudata()*, or *t_sndudata()* is issued, the function will return a TPROTO
9166 error.
- 9167 — **Specified value:** A value that will be known by a partner program; for
9168 example, a well-known transaction program name used by a server.
- 9169 — **Defined value, DYNAMIC_TPN:** An indication that the system should
9170 generate a TP name for the file descriptor.
- 9171 6. The values SYS_NET, SYS_LU and DYNAMIC_TPN may not be used as real
9172 values of the sna_netid, sna_lu or sna_tpn fields, respectively.

9173 I.2.3 Options

9174 Options are formatted according to the structure **t_opthdr** as described in Chapter 6. A
9175 transport provider compliant to this specification supports none, all, or any subset of the options
9176 defined in Section I.2.3.1.

9177 I.2.3.1 Connection-Mode Service Options

9178 The protocol level of all subsequent options is SNA.

9179 All options are association-related. Some may be negotiated in the XTI states T_IDLE and
9180 T_INCON, and all are read-only in all other states except T_UNINIT.

9181 **Options for Service Quality and Expedited Data**

9182 Table I-1 shows the SNA options that affect the quality of a connection and the transport service
 9183 level provided.

9184

9185

9186

9187

9188

9189

9190

9191

9192

9193

9194

9195

9196

9197

9198

9199

9200

9201

9202

9203

9204

9205

9206

9207

9208

9209

9210

9211

9212

9213

Option Name	Type of Option Value	Legal Option Value	Meaning
SNA_MODE	char	SNA_BATCH SNA_BATCHSC SNA_INTER SNA_INTERSC SNA_DEFAULT any user-defined SNA mode value	SNA mode, which controls the underlying class of service selected for the connection. The SNA mode is specified only by the active side of the connection. If not specified, the default mode is SNA_DEFAULT. The default mode characteristics may vary from system to system.
SNA_ALWAYS_PUSH	unsigned long	T_YES / T_NO	If T_YES, every <i>t_snd()</i> operation will cause the message to be sent immediately. If T_NO, the data from a <i>t_snd()</i> operation may be buffered and sent later. The transport user can set the T_PUSH flag on a <i>t_snd()</i> function call to cause the data to be sent immediately. Default value is T_NO. This option is primarily for programs migrated to SNA from other protocol stacks that always send data immediately. It allows them to request behaviour similar to that on the original provider. However, setting SNA_ALWAYS_PUSH to T_YES may affect its performance.

9214

Table I-1 SNA Options

9215 **I.2.4 Functions**

9216 *t_accept()* Since user data is not exchanged during connection establishment, the
9217 parameter *call-->udata.len* must be 0.

9218 *t_bind()* The *addr* field of the *t_bind* structure represents the local network-id-qualified
9219 LU name of the local logical unit and the transaction program name of the
9220 program issuing the *t_bind()* function.

9221 If the endpoint was bound in the passive mode (that is, *qlen* > 0) and the
9222 requested address has a null transaction program subfield, the function
9223 completes with the T_BADADDR error.

9224 *t_connect()* The *sndcall-->addr* specifies the network-ID-qualified LU name and transaction
9225 program name of the remote connection partner.

9226 An SNA transport provider allows more than one connection between the
9227 same address pair.

9228 Since user data cannot be exchanged during the connection establishment
9229 phase, *sndcall-->udata.len* must be set to 0. On return, *rcvcall-->udata.maxlen*
9230 should be set to 0.

9231 *t_getinfo()* In all states except T_DATAXFER, the function *t_getinfo()* returns in the
9232 parameter *info* the same information that was returned by *t_open()*. In
9233 T_DATAXFER state, however, the information returned may differ from that
9234 returned by *t_open()*, depending on whether the remote transport provider
9235 supports expedited data transfer. The fields of *info* are set as defined in the
9236 table below.

9237

9238

9239

9240

9241

9242

9243

9244

9245

9246

9247

Parameters	Before Call	After Call
fd	x	/
info-->addr	/	82
info-->options	/	x ¹
info-->tsdu	/	-1
info-->etsdu	/	-2 / 86 ²
info-->connect	/	-2
info-->discon	/	-2
info-->servtype	/	T_COTS_ORD
info-->flags	/	T_SNDZERO

9248

Table I-2 Fields for *info* Parameter

9249

Notes:

9250

1. x means an integral number greater than zero.

9251

2. Depending on the negotiation of expedited data transfer.

9252

9253

t_getprotaddr() The *boundaddr* value includes the transaction program name of the local
program.

9254

9255

The *peeraddr* value (if any) includes only the network-ID-qualified LU name of
the partner.

9256

9257

t_listen() The *call-->addr* structure contains the network-ID-qualified LU name of the
remote partner.

9258 `t_open()` The default characteristics returned by `t_open()` are shown in the table below.

9259
9260

9261
9262
9263
9264
9265
9266
9267
9268
9269
9270

Parameters	Before Call	After Call
name	x	/
oflag	x	/
info-->addr	/	82
info-->options	/	x ¹
info-->tsdu	/	-1
info-->etsdu	/	-2 / 86 ²
info-->connect	/	-2
info-->discon	/	-2
info-->servtype	/	T_COTS_ORD
info-->flags	/	T_SNDZERO

9271

Table I-3 Default Characteristics returned by `t_open()`

9272

Notes:

9273

1. x means an integral number greater than 0.

9274

2. Expedited data may or may not be supported by the local transport provider.

9275

9276 `t_rcv()`

9277
9278
9279

If expedited data arrives after part of a TSDU (logical record) has been retrieved, receipt of the remainder of the TSDU will be suspended until after 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.

9280 `t_rcvconnect()`

9281

Since no user data can be returned on `t_rcvconnect()`, the `call-->udata.len` should be set to 0 before the function is invoked.

9282 `t_rcvdis()`

9283

Since user data is not sent during disconnection, the value `discon-->udata.len` should be set to 0 before `t_rcvdis()` is called.

9284
9285

The following disconnect reason codes are valid for any implementation of an SNA transport provider under XTI:

9286
9287
9288
9289
9290

```
#define SNA_CONNECTION_SETUP_FAILURE.  
#define SNA_USER_DISCONNECT  
#define SNA_SYSTEM_DISCONNECT  
#define SNA_TIMEOUT  
#define SNA_CONNECTION_OUTAGE
```

9291

These definitions should be included in `<xti.h>`.

9292 `t_snd()`

9293
9294
9295

Unless the SNA_ALWAYS_PUSH option is set to T_YES or the T_PUSH flag on the `t_snd()` function is set, the SNA transport provider may collect data in a send buffer until it accumulates a sufficient amount for transmission. The amount of data that is accumulated can vary from one connection to another.

9296
9297
9298
9299

In order to take advantage of the performance improvement that SNA buffering offers, the XTI user must set the SNA_ALWAYS_PUSH option to T_NO (Default is T_YES). If this option is not set to T_NO, a push will be done for every `t_snd()` and the T_PUSH flag will have no effect.

9300 `t_snddis()`

9301

Since no user data is sent during a disconnect operation, `call-->udata.len` should be set to 0 before the call to `t_snddis()`.

9311 I.3 Mapping XTI to SNA Transport Provider

9312 This section presents the mapping of XTI functions to Full Duplex (FDX) LU 6.2.

9313 First, several flow diagrams are given to illustrate the function of the XTI Mapper. Following
9314 this, mapping tables are given that show the FDX LU 6.2 verbs that the XTI Mapper needs to
9315 generate for each XTI function. For each XTI function that maps to a FDX verb, an additional
9316 table is referenced that gives the mappings of each parameter. Finally, a table shows mapping of
9317 LU 6.2 FDX return codes to XTI events.

9318 The use of FDX LU 6.2 verbs in this section is for illustrative purposes only and is analogous to
9319 OSI's use of service primitives, that is, as a way to explain the semantics provided by the
9320 protocol. The FDX LU6.2 verbs are used only to help in understanding the SNA protocol, and
9321 are not a required part of an implementation.

9322 **I.3.1 General Guidelines**

9323 General guidelines for mapping XTI to an SNA transport provider are listed below:

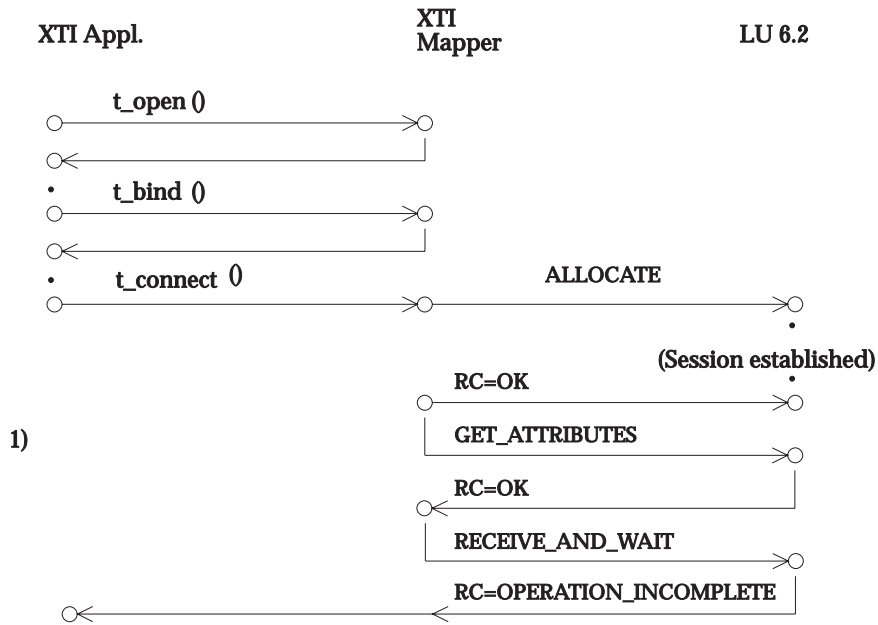
- 9324 • In the following flow diagrams, notice that the XTI mapper always has a
 9325 RECEIVE_AND_WAIT posted. This is done so that when data comes into the SNA transport
 9326 provider, the XTI mapper is able to set an event indicator. Then, when a T_LOOK is issued,
 9327 the XTI application can be informed that there is data to be received.
- 9328 • The XTI mapper keeps a table that maps an XTI fd to a RESOURCE(variable) on the FDX
 9329 verbs.
- 9330 • In this section, we assume the XTI mapper will be using the FDX LU 6.2 basic conversation
 9331 verb interface. The following table Table I-4 gives an explanation for each FDX verb that is
 9332 used in these mappings.

9333 **Table I-4 FDX LU 6.2 Verb Definitions**

FDX Verb	Description
ALLOCATE	Allocates a conversation between the local transaction program and a remote (partner) transaction program.
DEALLOCATE	DEALLOCATE with TYPE(FLUSH) closes the local program's send queue. Both the local and remote program must close their send queues independently. DEALLOCATE with TYPE(ABEND_PROG) is an abrupt termination that will close both sides of the conversation simultaneously.
FLUSH	Flushes the local LU's send buffer.
GET_ATTRIBUTES	Returns information pertaining to the specified conversation.
GET_TP_PROPERTIES	Returns information pertaining to the transaction program issuing the verb.
RECEIVE_ALLOCATE	Receives a new conversation with a partner transaction program that issued ALLOCATE.
RECEIVE_AND_WAIT	Waits for data to arrive on the specified conversation and then receives the data. If data is already available, the program receives it without waiting.
RECEIVE_EXPEDITED_DATA	Receives data sent by the remote transaction program in an expedited manner, via the SEND_EXPEDITED_DATA verb.
SEND_DATA	Sends data to the remote transaction program.
SEND_EXPEDITED_DATA	Sends data to the remote transaction program in an expedited manner. This means that it may arrive at the remote transaction program before data sent earlier via a send queue verb; for example, SEND_DATA.
WAIT_FOR_COMPLETION	Waits for posting to occur on one or more non-blocking operations represented in the specified wait objects. Posting of a non-blocking operation occurs when the LU has completed the associated non-blocking verb and filled all the return values.

9365 **I.3.2 Flows Illustrating Full Duplex Mapping**

9366 The following diagrams show mappings from the XTI function calls for active connection
 9367 establishment to SNA verb sequences. The first Figure I-1 is used for blocking XTI calls; the
 9368 second Figure I-2 is used for non-blocking calls.
 9369

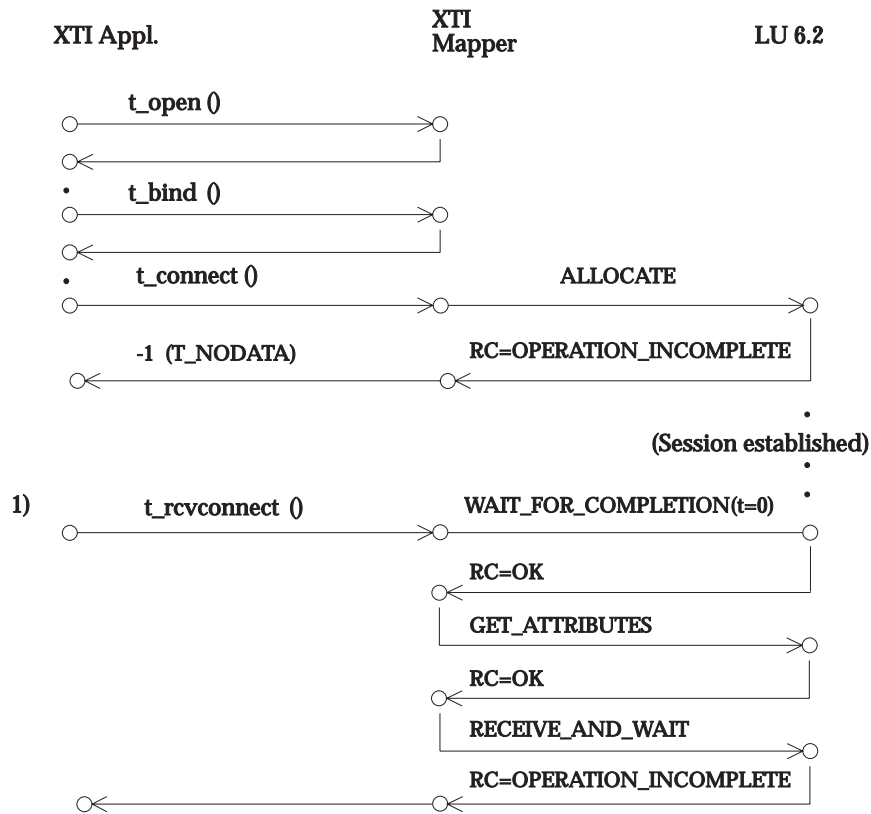


9370 **Figure I-1** Active Connection Establishment, Blocking Version (1 of 2)

9371 **Annotations**

- 9372 1. `GET_ATTRIBUTES` is issued after the session is established and before the return for the
 9373 `t_connect`. This is only done if the mode name, or partner LU name are required on the
 9374 return to `t_connect`. This would be indicated by a non-zero value in either the `rcvcall--`
 9375 `>addr.buf` or `eercvcall-->opt.buf` fields on `t_connect`.

9376



9377

Figure I-2 Active Connection Establishment, Non-blocking Version (2 of 2)

9378

Annotations

9379
9380
9381
9382

1. The XTI application will issue a *t_rcvconnect* as a poll to see if the *t_connect* has completed. The *t_rcvconnect* will cause a *WAIT_FOR_COMPLETION*, with time=0, to be issued. The *WAIT_FOR_COMPLETION* will check on the wait object from the previous non-blocking *ALLOCATE*.

9383
9384

When the *t_connect* has completed successfully a *GET_ATTRIBUTES* is issued if the mode name, or partner LU name are required on the return of the *t_rcvconnect*.

9385
9386

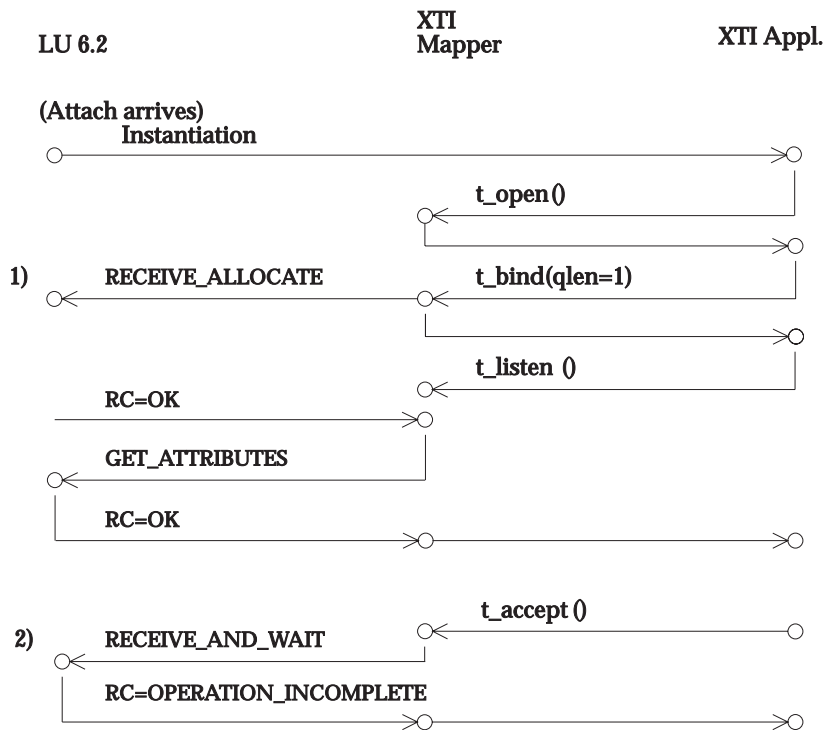
After the *GET_ATTRIBUTES*, a non-blocking *RECEIVE_AND_WAIT* is issued to post a receive for any incoming data.

9387 The next three diagrams show possible mappings of SNA Attach processing for an incoming
 9388 connection to the series of XTI calls on the passive side of a connection.

9389 The first Figure I-3 uses the native SNA instantiation mechanism; that is, programs are
 9390 instantiated when the connection request arrives. This requires that the TP name (that is, the
 9391 XTI application name) is known as part of the LU definition. This is **before** the *t_bind* is issued.

9392 The second Figure I-4 is a blocking use of the interface, where the SNA transport provider allows
 9393 a connection request to be received by an existing program. This model, although not described
 9394 in the architecture, is supported by many SNA products.

9395 The third Figure I-5 is a non-blocking use of the interface, where the SNA transport provider
 9396 allows a connection request to be received by an existing program. This model is described as
 9397 part of the FDX architecture.
 9398

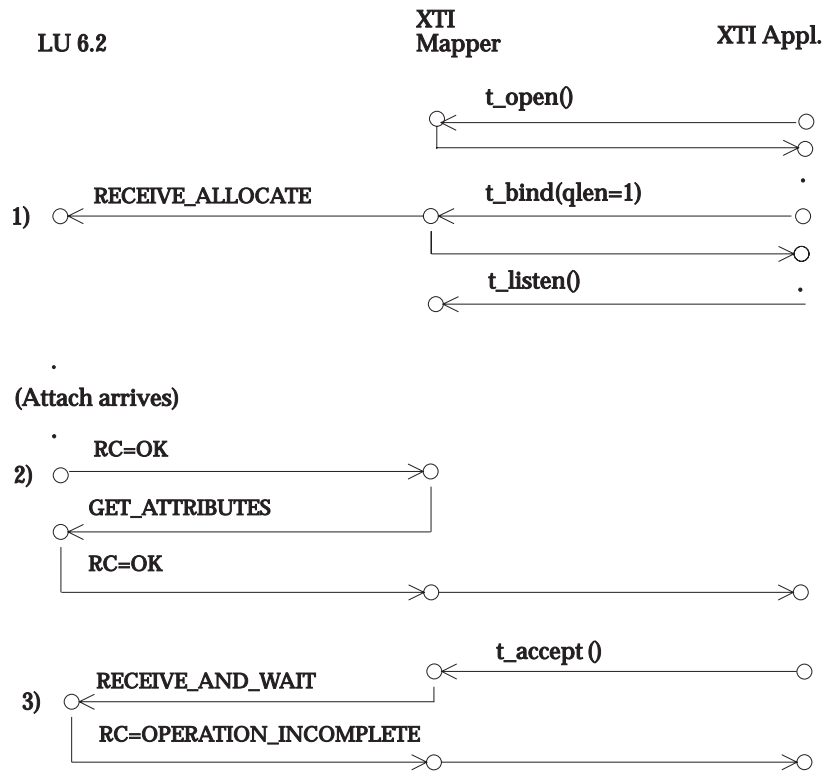


9399 **Figure I-3** Passive Connection Establishment, Instantiation Version (1 of 3)

9400 **Annotations**

- 9401 1. If *qlen* in *t_bind* is > 0, a *RECEIVE_ALLOCATE* will be issued for each connection request
 9402 that can be queued. When the *RECEIVE_ALLOCATE* completes successfully a
 9403 *GET_ATTRIBUTES* is issued only if the mode name, or partner LU name are required on
 9404 the return to *t_listen*.
- 9405 2. The *t_accept* will cause a *RECEIVE_AND_WAIT* to be issued. The *RECEIVE_AND_WAIT*
 9406 is issued to post a receive for any incoming data.

9407



9408

Figure I-4 Passive Connection Establishment, Blocking Version (2 of 3)

9409

Annotations

9410

1. The *t_bind* will cause a blocking RECEIVE_ALLOCATE to be issued for each connection request that can be queued.

9411

9412

2. When the RECEIVE_ALLOCATE completes successfully a GET_ATTRIBUTES is issued only if the mode name, or partner LU name are required on the return to *t_listen*.

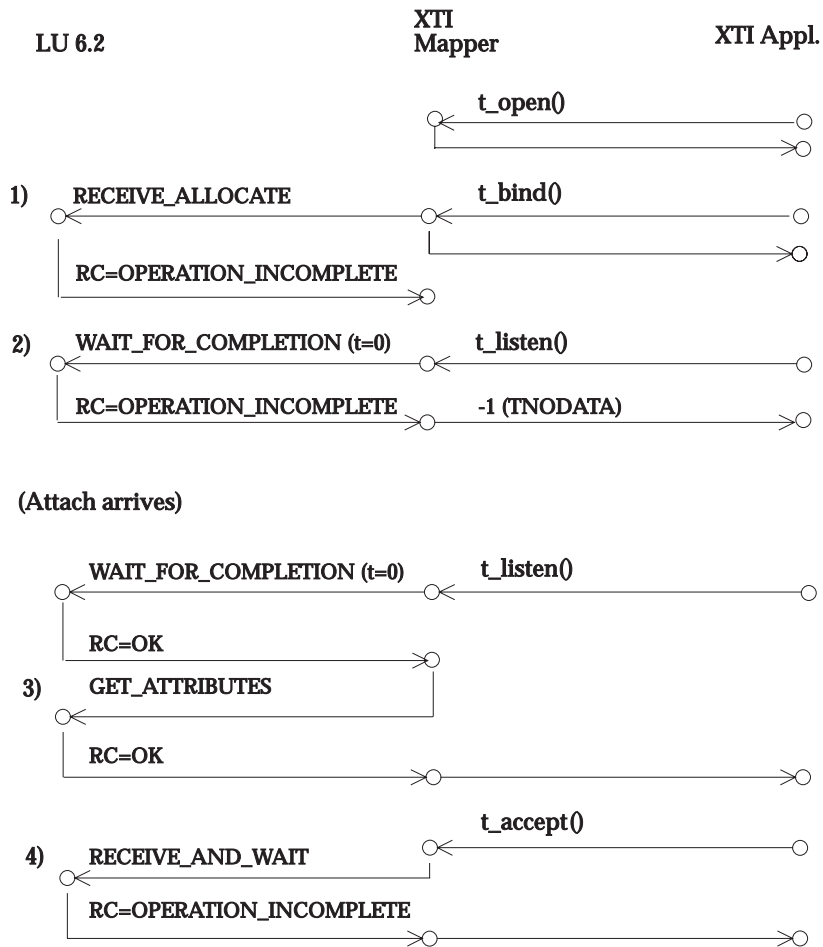
9413

9414

3. The *t_accept* will cause a RECEIVE_AND_WAIT to be issued. The RECEIVE_AND_WAIT is issued to post a receive for any incoming data.

9415

9416



9417

Figure I-5 Passive Connection Establishment, Non-blocking Version (3 of 3)

9418

Annotations

9419
9420

1. The `t_bind` will cause a non-blocking `RECEIVE_ALLOCATE` to be issued for each connection request that can be queued.

9421
9422
9423
9424
9425
9426

2. A `t_listen` is used as a poll to see if a connect request has been received. The `t_listen` will cause a `WAIT_FOR_COMPLETION`, with `time=0`, to be issued. The `WAIT_FOR_COMPLETION` will check on the wait object from the previous non-blocking `RECEIVE_ALLOCATE`. In this example, when the first `t_listen` is issued, the `RECEIVE_ALLOCATE` is still outstanding; but the `RECEIVE_ALLOCATE` has completed before the second `t_listen` is issued.

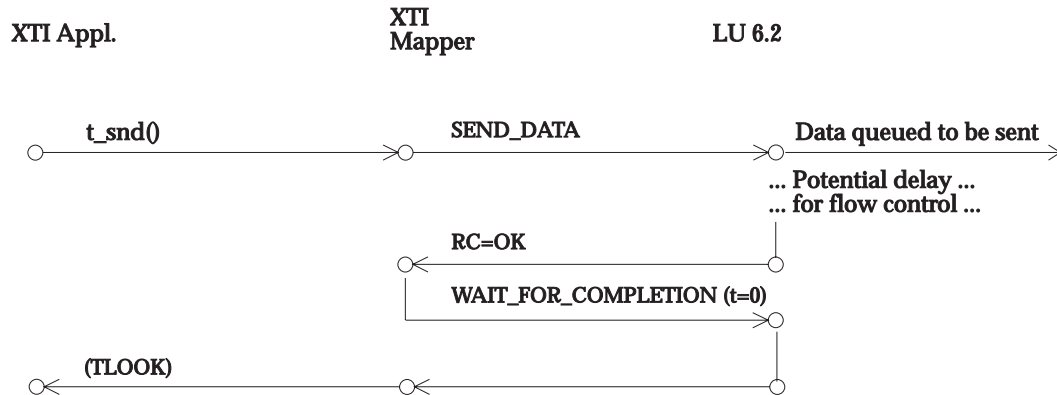
9427
9428
9429

3. When the `WAIT_FOR_COMPLETION` indicates that the `RECEIVE_ALLOCATE` has completed successfully, a `GET_ATTRIBUTES` is issued only if the mode name, or partner LU name are required on the return to `t_listen`.

9430
9431

4. The `t_accept` will cause a `RECEIVE_AND_WAIT` to be issued. The `RECEIVE_AND_WAIT` is issued to post a receive for any incoming data.

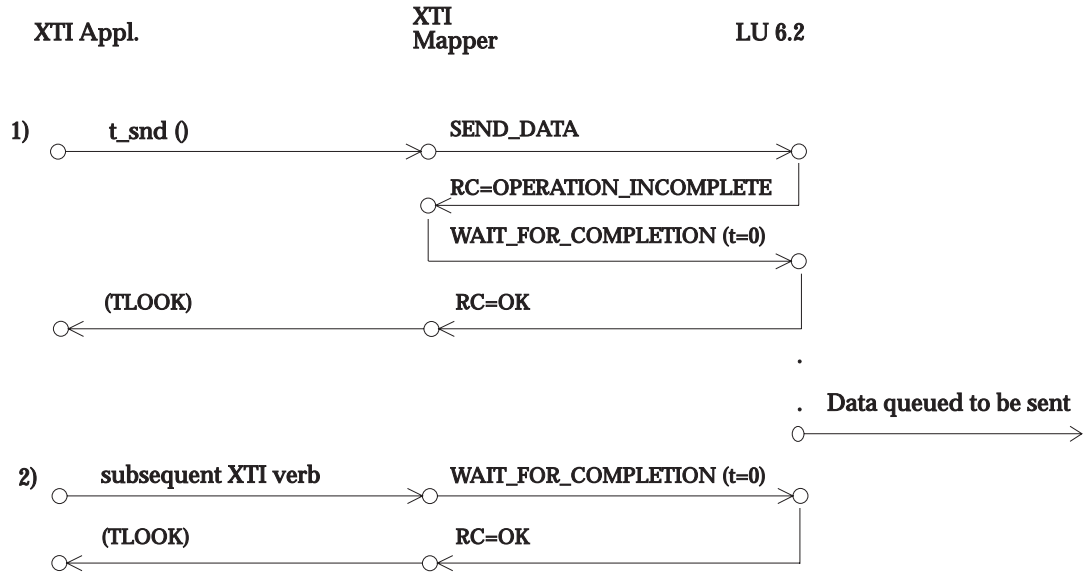
9432 The next diagram, Figure I-6, shows the mapping for the blocking XTI *t_snd()* call. The diagram
 9433 after this, Figure I-7, shows the non-blocking mapping of the XTI *t_snd()* call.
 9434



9435 **Figure I-6** XTI Function to LU 6.2 Verb Mapping: Blocking *t_snd*

- 9436 1. The blocking *t_snd* will cause a blocking SEND_DATA to be issued. This will block until
 9437 the LU accepts and queues all the data being sent.
- 9438 If EXPEDITED=YES, the mapper will issue a SEND_EXPEDITED_DATA verb rather than
 9439 the SEND_DATA.
- 9440 2. When the SEND_DATA returns, a WAIT_FOR_COMPLETION, with time=0, is issued to
 9441 see if the wait object for any outstanding non-blocking LU 6.2 verbs have been posted. At
 9442 a minimum, there will be an outstanding RECEIVE_AND_WAIT, waiting for any incoming
 9443 data, that needs to be checked. If any wait objects have been posted, the return code on the
 9444 *t_snd* is set to TLOOK. This will inform the XTI application to issue a *t_look* to see what
 9445 has been posted.

9446



9447

Figure I-7 XTI Function to LU 6.2 Verb Mapping: Non-blocking t_snd

9448
9449
9450

1. The XTI mapper needs to either accept all the data being sent, or none of it. In this case, all the data is accepted, thus the non-blocking *t_snd* causes a non-blocking SEND_DATA to be issued.

9451
9452

The XTI mapper then needs to issue a WAIT_FOR_COMPLETION to see if any other blocking LU 6.2 verbs have completed. This case is not shown in this diagram.

9453
9454

If EXPEDITED=YES, the mapper will issue a SEND_EXPEDITED_DATA verb rather than the SEND_DATA.

9455
9456
9457
9458
9459
9460

2. When a subsequent XTI verb is issued (for example, *t_rcv* or *t_send*), a WAIT_FOR_COMPLETION, with time=0, is issued to see if the wait object for any outstanding non-blocking LU 6.2 verbs have been posted. In this case, one of the wait objects will be the one associated with the non-blocking SEND_DATA. If any wait objects have been posted, the return code on the *t_snd* is set to TLOOK. This will inform the XTI application to issue a *t_look* to see what has been posted.

9461
9462

There may be an additional LU 6.2 verb issued due to the *subsequent XTI* verb that was issued. This is not shown in the above diagram.

9463
9464
9465

The next diagram, Figure I-8, shows the mapping for blocking XTI receive call, and the diagram after this, Figure I-9, shows the mapping for non-blocking XTI receive call.

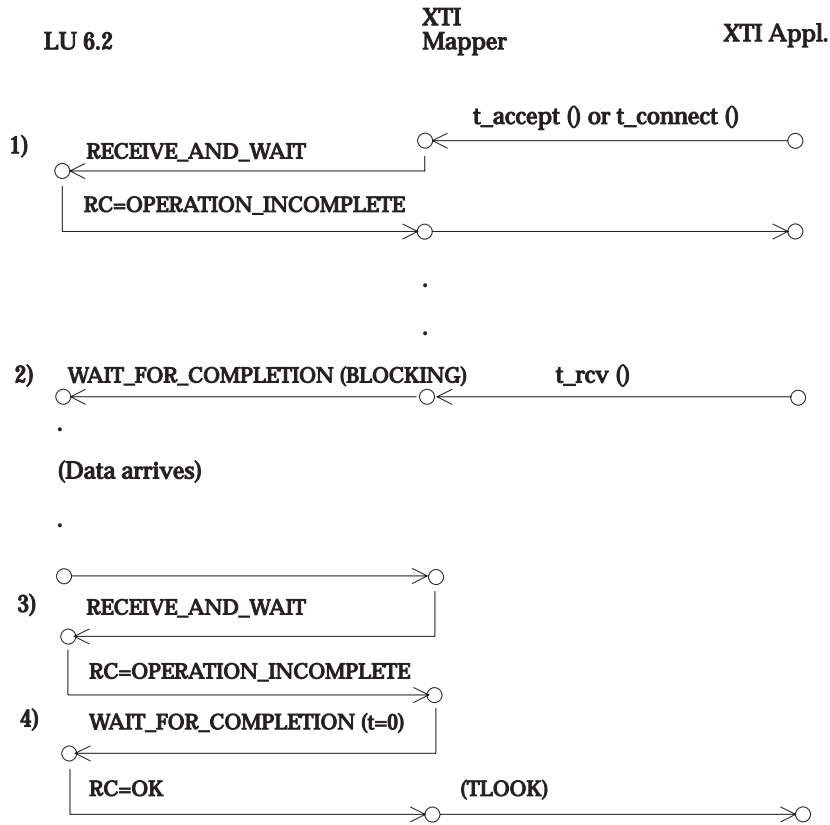


Figure I-8 XTI Function to LU 6.2 Verb Mapping: Blocking t_rcv

9466

1. There is always an outstanding non-blocking RECEIVE_AND_WAIT, this is true whether the XTI application is using blocking or non-blocking mode. This is to post a receive for any incoming data.

9467
9468
9469

In this diagram, the outstanding RECEIVE_AND_WAIT was issued when the connection was setup. This could be as a result of either a t_accept or t_connect.

9470
9471

2. When the XTI issues a blocking t_rcv, the XTI mapper will issue a blocking WAIT_FOR_COMPLETION to wait on the wait object associated with the outstanding RECEIVE_AND_WAIT. This will block until data is received on this connection.

9472
9473
9474

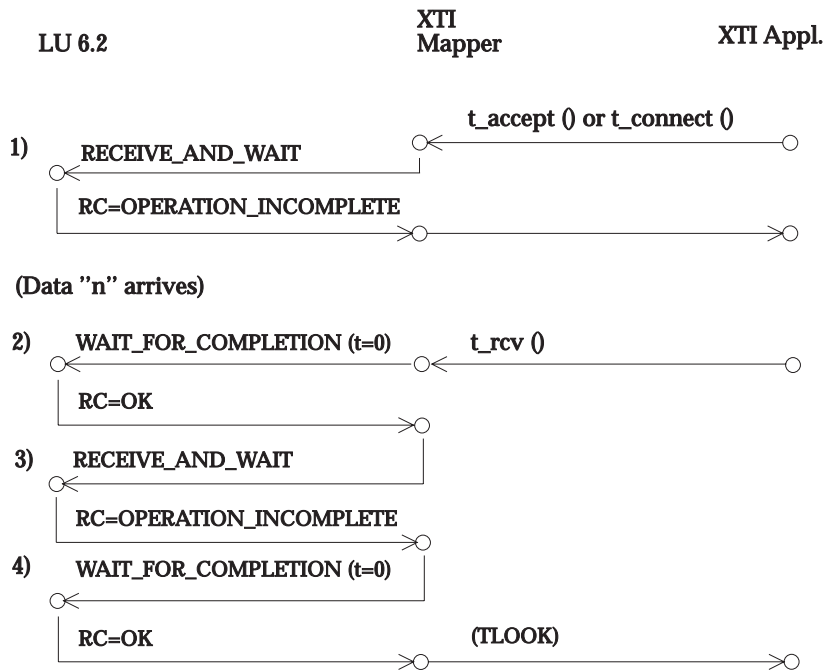
3. When data is received, the mapper needs to issue a non-blocking RECEIVE_AND_WAIT to replace the one that just completed.

9475
9476

4. Issue a WAIT_FOR_COMPLETION, with time=0, to see if any other outstanding wait objects have been posted.

9477
9478

9479



9480

Figure I-9 Mapping from XTI Calls to LU 6.2 Verbs (Passive side)

9481

1. There is always an outstanding non-blocking RECEIVE_AND_WAIT, this is true whether the XTI application is using blocking or non-blocking mode. This is to post a receive for any incoming data.

9482

9483

9484

In this diagram, the outstanding RECEIVE_AND_WAIT was issued when the connection was setup. This could be as a result of either a *t_accept* or *t_connect*.

9485

9486

2. When the XTI application issues a non-blocking *t_rcv*, the XTI mapper will issue a WAIT_FOR_COMPLETION, with T=0, to see if the wait object for the outstanding RECEIVE_AND_WAIT has been posted. When the wait object has been posted, the XTI mapper needs to pass the data to the XTI application buffer.

9487

9488

9489

9490

It is possible that the amount of incoming data in the XTI mapper buffer is more than the XTI application stated on the *t_rcv*. In this case, the XTI Mapper will set the TMORE flag. Then, when the next *t_rcv* is issued, the remaining data will be passes to the XTI application **BEFORE** issuing the WAIT_FOR_COMPLETION to check the wait object on the outstanding RECEIVE_AND_WAIT.

9491

9492

9493

9494

9495

3. When data is received, the mapper needs to issue a non-blocking RECEIVE_AND_WAIT to replace the one that just completed.

9496

9497

4. Issue a WAIT_FOR_COMPLETION, with time=0, to see if any other outstanding wait objects have been posted.

9498

9499 **I.3.3 Full Duplex Mapping**

9500 The following table shows the mapping from XTI function calls to full duplex LU 6.2 verbs.

9501 **Table I-5 XTI Mapping to LU 6.2 Full Duplex Verbs**

9502	XTI Function	SNA FDX LU6.2 verb	Comments
9503	<i>t_accept()</i>	RECEIVE_AND_WAIT	User data is not exchanged during connection establishment.
9504			Refer to Table I-7 on page 314.
9505			
9506	<i>t_alloc()</i>	Local	
9507	<i>t_bind()</i>	If qlen>0: RECEIVE_ALLOCATE for each connection request that can be queued.	Refer to Table I-8 on page 315.
9508		Optionally: DEFINE_TP	
9509		With the instantiation model, the TP name (that is, XTI application name) must be known by the LU before the TP can be instantiated.	
9510		This is prior to the <i>t_bind</i> being issued. (Refer to Figure I-3 on page 305.)	
9511			
9512			
9513			
9514			
9515			
9516			
9517			
9518			
9519	<i>t_close()</i>	If connection still up issue DEALLOCATE TYPE(ABEND)	May be a delay if XTI_LINGER option activated with non-zero linger value.
9520			Refer to Table I-9 on page 315.
9521			
9522			
9523	<i>t_connect()</i>	ALLOCATE RETURN_CONTROL	Refer to Table I-10 on page 316.
9524		GET_ATTRIBUTES	
9525		RECEIVE_AND_WAIT	
9526	<i>t_error()</i>	Local	
9527	<i>t_free()</i>	Local	
9528	<i>t_getinfo()</i>	Local	
9529	<i>t_getprotaddr()</i>	GET_TP_PROPERTIES to get OWN_FULLY_QUALIFIED_LU_NAME and OWN_TP_NAME	The partner's TP name must be learned by some mechanism other than XTI services. In connectionless mode, there is no partner name.
9530		GET_ATTRIBUTES to get PARTNER_FULLY_QUALIFIED_LU_NAME	Refer to Table I-11 on page 318.
9531			
9532			
9533			
9534			
9535	<i>t_getstate()</i>	Local	
9536	<i>t_listen()</i>	WAIT_FOR_COMPLETION	Refer to Table I-12 on page 319.
9537	<i>t_look()</i>	WAIT_FOR_COMPLETION	
9538	<i>t_open()</i>	Local	<i>t_open</i> sets blocking mode (that is, blocking or non-blocking)
9539			
9540	<i>t_optmgmt()</i>	GET_ATTRIBUTES	To get Mode name
9541			Refer to Table I-13 on page 319.

9542	XTI Function <i>t_rcv()</i>	SNA FDX LU6.2 verb WAIT_FOR_COMPLETION	Comments Refer to Table I-14 on page 320.
9543		RECEIVE_AND_WAIT	
9544		[RECEIVE_EXPEDITED_DATA]	
9545		WAIT_FOR_COMPLETION	
9546	<i>t_rcvconnect()</i>	WAIT_FOR_COMPLETION	Refer to Figure I-2 on page 304.
9547		GET_ATTRIBUTES	
9548		RECEIVE_AND_WAIT	
9549	<i>t_rcvdis()</i>	Local	Event caused by DEALLOCATE_ABEND_* or RESOURCE_FAILURE_* return code on any verb
9550			
9551			
9552			
9553	<i>t_rcvrel()</i>	Local	Event caused by DEALLOCATE_NORMAL return code on RECEIVE_* verb
9554			
9555			
9556	<i>t_snd()</i>	SEND_DATA (expedited data)	Every <i>t_snd</i> causes a SEND_DATA to be issued - even if T_MORE set. If T_MORE is set, the LL continuation bit is set. A zero-length TSDU causes the following LL to be sent: hex 0002. This can be used to turn off the LL continuation set on the previous send. Refer to Table I-16 on page 322.
9557		[FLUSH]	
9558		[SEND_EXPEDITED_DATA]	
9559			
9560			
9561			
9562			
9563			
9564			
9565			
9566	<i>t_snddis()</i>	DEALLOCATE TYPE(ABEND_PROG)	Takes down both directions of the connection Refer to Table I-17 on page 323.
9567			
9568	<i>t_sndrel()</i>	DEALLOCATE TYPE(FLUSH)	Takes down send direction of conversation only. Refer to Table I-19 on page 323.
9569			
9570			
9571	<i>t_sndudata()</i>	SEND_DATA on datagram server conversation	Refer to Table I-19 on page 323.
9572			
9573	<i>t_streerror()</i>	local	
9574	<i>t_sync()</i>	Local	
9575	<i>t_unbind()</i>	Local	

9578 I.3.3.1 Parameter Mappings

9579

Table I-6 Relation Symbol Description

9580

9581

9582

9583

9584

9585

9586

9587

Relation Symbol	Meaning
Used Locally	Value is used locally by XTI Mapper
Created Locally	XTI Mapper creates the value
Constant	Only one value is acceptable in this field. It is an error condition if any other value is passed.
<---	XTI Application parameter maps directly to FDX Verb parameter.
--->	FDX Verb parameter maps directly to XTI Application parameter.

9588

Table I-7 *t_accept* <--> FDX Verbs and Parameters

9589

9590

9591

9592

9593

9594

9595

9596

9597

9598

9599

9600

9601

9602

9603

9604

9605

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_accept		RECEIVE_AND_WAIT
Input		
fd	Used Locally	
resfd	---->	RESOURCE(variable)
call-->addr.len	Used Locally	
call-->addr.buf	Used Locally	
call-->opt.len	Used Locally	
call-->opt.buf	Used Locally	
call-->udata.len	Constant	=0
call-->udata.buf	Constant	=nullptr
	Created Locally	LENGTH(variable)
	Created Locally	FILL(addr of local bufr)
	Constant	WAIT_OBJECT(BLOCKING)
Output		
t_errno	<----	RETURN_CODE(variable)

9606

Table I-8 *t_bind* <--> FDX Verbs and Parameters

9607

9608

9609

9610

9611

9612

9613

9614

9615

9616

9617

9618

9619

9620

9621

9622

9623

9624

9625

9626

9627

9628

9629

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_bind with qlen>0		RECEIVE_ALLOCATE
Input		
fd	Used Locally	
req-->addr.len	Used Locally	
req-->addr.buf	<----	LOCAL_LU_NAME(variable) TP_NAME(variable)
req-->qlen	Used Locally	>0, RECEIVE_ALLOCATE issued for each connect request that can be queued.
ret-->addr.maxlen	Used Locally	
	Constant	RETURN_CONTROL (WHEN_ALLOCATE_RECEIVED)
	Constant	SCOPE(ALL)
	Created Locally	WAIT_OBJECT(BLOCKING)
Output		
ret-->addr.len	Created Locally	
ret-->addr.buf	---->	LOCAL_LU_NAME(variable) TP_AL_LU_NAME(variable)
ret-->addr.qlen	Created Locally	
	Used Locally	RESOURCE(variable)
t_errno	---->	RETURN_CODE(variable)

9630

Table I-9 *t_close* <--> FDX Verbs and Parameters

9631

9632

9633

9634

9635

9636

9637

9638

9639

9640

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_close		DEALLOCATE
If connection is still in T_DATAXFER state		
Input		
fd	---->	RESOURCE(variable)
	Constant	TYPE(ABEND_PROG)
Output		
t_errno	<----	RETURN_CODE(variable)

9641

Table I-10 *t_connect* <--> FDX Verbs and Parameters

9642

9643

9644

9645

9646

9647

9648

9649

9650

9651

9652

9653

9654

9655

9656

9657

9658

9659

9660

9661

9662

9663

9664

9665

9666

9667

9668

9669

9670

9671

9672

9673

9674

9675

9676

9677

9678

9679

9680

9681

9682

9683

9684

9685

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_connect		ALLOCATE RETURN_CONTROL
Input		
fd		
sndcall-->addr.len	Used Locally	
sndcall-->addr.buf	---->	LU_NAME(), TP_NAME()
sndcall-->opt.len	Used Locally	
sndcall-->opt.buf	---->	MODE_NAME()
sndcall-->udata.len	Constant	=0, user data not allowed
sndcall-->udata.buf	Constant	=nullptr
rcvcall-->addr.maxlen	Used Locally	
rcvcall-->addr.buf	Used Locally	
rcvcall-->opt.maxlen	Constant	=0, user data not allowed
rcvcall-->opt.buf	Used Locally	
rcvcall-->udata.maxlen	Constant	=0, user data not allowed
rcvcall-->udata.buf	Constant	=nullptr
	Constant	TYPE(FULL_DUPLEX_BASIC_CONV)
	Created Locally	RETURN_CODE (WHEN_SESSION_FREE) If platform does not support this tower (Tower 205), use (WHEN_SESSION_ALLOCATED).
	Created Locally	WAIT_OBJECT(BLOCKING) if blocking WAIT_OBJECT(VALUE(variable)) if non-blocking
Output		
	Used Locally	RESOURCE(variable)
t_errno	<----	
t_connect		GET_ATTRIBUTES
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
Output		
rcvcall-->addr.len	<----	PARTNER_FULLY_QUALIFIED_ LU_NAME(variable)
rcvcall-->addr.buf	<-----	PARTNER_FULLY_QUALIFIED_ LU_NAME(variable)
rcvcall-->opt.len	<----	MODE_NAME(variable)
rcvcall-->opt.buf	<----	MODE_NAME(variable)
t_errno	<----	RETURN_CODE(variable)
t_connect		RECEIVE_AND_WAIT
Input		

9686
 9687
 9688
 9689
 9690
 9691
 9692
 9693

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
fd	Used Locally	
	Created Locally	RESOURCE(variable)
Output		
	Created Locally	LENGTH(variable)
	Created Locally	FILL(addr of local bufr)
	Constant	WAIT_OBJECT(BLOCKING)

9694

Table I-11 *t_getprocaddr* <--> FDX Verbs and Parameters

9695

9696

9697

9698

9699

9700

9701

9702

9703

9704

9705

9706

9707

9708

9709

9710

9711

9712

9713

9714

9715

9716

9717

9718

9719

9720

9721

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_getprotaddr		GET_ATTRIBUTES to get partner LU name
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
boundaddr-->maxlen	Used Locally	
boundaddr-->addr.buf	Used Locally	
peeraddr-->maxlen	Used Locally	
peeraddr-->addr.buf	Used Locally	
Output		
peeraddr-->addr.len	Created Locally	
buf(peeraddr-->addr.buf)	<----	PARTNER_FULLY_QUALIFIED_LU_NAME(variable)
t_errno	<----	RETURN_CODE(variable)
t_getprotaddr		GET_TP_PROPERTIES to get local TP name
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
Output		
boundaddr-->addr.len	Created Locally	
buf(boundaddr-->addr.buf)	<----	OWN_FULLY_QUALIFIED_LU_NAME(variable)
		OWN_TP_NAME(variable)
t_errno	<----	RETURN_CODE(variable)

9722

Table I-12 *t_listen* <--> FDX Verbs and Parameters

9723

9724

9725

9726

9727

9728

9729

9730

9731

9732

9733

9734

9735

9736

9737

9738

9739

9740

9741

9742

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_listen		WAIT_FOR_COMPLETION
Input		
	Created Locally	WAIT_OBJECT_LIST(variable)
	Constant	TIMEOUT(VALUE(variable=0))
Output		
t_errno	<----	RETURN_CODE(variable)
	Used Locally	STATUS_LIST(variable)
t_listen		GET_ATTRIBUTES
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
Output		
call-->addr.len	<-----	PARTNER_FULLY_QUALIFIED_LU_NAME(variable)
bufr&larrow.(call-->addr.bufr)	<-----	PARTNER_FULLY_QUALIFIED_LU_NAME(variable)
call-->opt.len	<-----	MODE_NAME(variable)
bufr&larrow.(call-->opt.bufr)	<-----	MODE_NAME(variable)

9743

Table I-13 *t_optmgmt* <--> FDX Verbs and Parameters

9744

9745

9746

9747

9748

9749

9750

9751

9752

9753

9754

9755

9756

9757

9758

9759

9760

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_optmgmt		GET_ATTRIBUTES
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
req-->opt.maxlen	Used Locally	
req-->opt.len	Used Locally	
req-->opt.bufr	Used Locally	
req-->opt.flags	Used Locally	
ret-->opt.maxlen	Used Locally	
ret-->opt.bufr	Used Locally	
Output		
ret-->opt.len	<-----	MODE_NAME(variable)
ret-->opt.bufr	<-----	MODE_NAME(variable)
ret-->flags	Created Locally	
t_errno	<-----	RETURN_CODE(variable)

9761

Table I-14 *t_rcv* <--> FDX Verbs and Parameters

9762

9763

9764

9765

9766

9767

9768

9769

9770

9771

9772

9773

9774

9775

9776

9777

9778

9779

9780

9781

9782

9783

9784

9785

9786

9787

9788

9789

9790

9791

9792

9793

9794

9795

9796

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_rcv		RECEIVE_AND_WAIT
A RECEIVE_AND_WAIT has been issued prior to this t_rcv. The data received on this RECEIVE_AND_WAIT will be returned to the XTI application via the t_rcv.		
Before the return to the t_rcv, the mapper will issue another RECEIVE_AND_WAIT to post a receive for any incoming data.		
Input		This is the RECEIVE_AND_WAIT that will be issued before the return to t_rcv.
fd	Used Locally	
	Created Locally	RESOURCE(variable)
nbytes	Used Locally	
	Created Locally	LENGTH(variable)
	Created Locally	FILL(XTI mapper buffer)
	Created Locally	WAIT_OBJECT(VALUE(variable))
t_rcv		RECEIVE_AND_WAIT
Output		These are fields from the previously issued RECEIVE_AND_WAIT
buf	<----	Data from FILL buffer
Return Value for function	<----	LENGTH(variable)
flags	Created Locally	WHAT_RECEIVED(variable)
• T_MORE		If there is expedited data to be received, a RECEIVE_EXPEDITED_DATA verb will be issued to receive it.
• T_EXPEDITED=NO/YES		
t_errno	<----	RETURN_CODE
t_rcv		WAIT_FOR_COMPLETION
Input		
	Created Locally	WAIT_OBJECT_LIST(variable)
	Constant	TIMEOUT(VALUE(variable=0))
Output		
errno	<----	RETURN_CODE(variable)
T_LOOK	Used Locally	STATUS_LIST(variable)

9797

Table I-15 *t_rcvconnect* <--> FDX Verbs and Parameters

9798

9799

9800

9801

9802

9803

9804

9805

9806

9807

9808

9809

9810

9811

9812

9813

9814

9815

9816

9817

9818

9819

9820

9821

9822

9823

9824

9825

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_rcvconnect		GET_ATTRIBUTES
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
Output		
call-->addr.len	<----	PARTNER_FULLY_QUALIFIED_LU_NAME(variable)
call-->addr.buf	<----	PARTNER_FULLY_QUALIFIED_LU_NAME(variable)
call-->opt.len	<----	MODE_NAME(variable)
call-->opt.buf	<----	MODE_NAME(variable)
t_errno	<----	RETURN_CODE(variable)
t_rcvconnect		RECEIVE_AND_WAIT
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
	Created Locally	LENGTH(variable)
	Created Locally	FILL(addr of local bufr)
	Constant	WAIT_OBJECT(VALUE(variable))
t_rcvconnect		WAIT_FOR_COMPLETION
Input		
	Created Locally	WAIT_OBJECT_LIST(variable)
	Constant	TIMEOUT(VALUE(variable=0))
Output		
t_errno	<----	RETURN_CODE(variable)
T_LOOK	Used Locally	STATUS_LIST(variable)

9826

Table I-16 *t_snd* <--> FDX Verbs and Parameters

9827

9828

9829

9830

9831

9832

9833

9834

9835

9836

9837

9838

9839

9840

9841

9842

9843

9844

9845

9846

9847

9848

9849

9850

9851

9852

9853

9854

9855

9856

9857

9858

9859

9860

9861

9862

XTI Function and Parameters	<--Relation-->	FDX Verb & Parameter
t_snd()		SEND_DATA
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
buf	---->	DATA(variable)
nbytes	---->	LENGTH(variable)
flags	---->	LL continuation bit
<ul style="list-style-type: none"> • T_EXPEDITED=NO • T_MORE • T_FLUSH 		
	Created Locally	WAIT_OBJECT(BLOCKING) if blocking
		WAIT_OBJECT(VALUE(variable)) if non-blocking
Output		
(TLOOK)	<----	EXPEDITED_DATA_RECEIVED
t_errno	<----	RETURN_CODE
t_snd()		SEND_EXPEDITED_DATA
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
buf	---->	DATA(variable)
nbytes	---->	LENGTH(variable)
flags	---->	LL continuation bit
<ul style="list-style-type: none"> • T_EXPEDITED=YES • T_MORE • T_FLUSH 		FLUSH Verb
	Created Locally	WAIT_OBJECT(BLOCKING) if blocking
WAIT_OBJECT(VALUE(variable)) if non-blocking		
Output		
(TLOOK)	<----	EXPEDITED_DATA_RECEIVED
t_errno	<----	RETURN_CODE

9863 **Table I-17** *t_snddis* (Existing Connection) <--> FDX Verbs and Parameters

9864 XTI Function and Parameters	9865 <--Relation-->	9866 FDX Verb & Parameter
9866 t_snddis()		DEALLOCATE
9867 For existing connection		
9868 Input		
9869 fd	Used Locally	
9870	Created Locally	RESOURCE(variable)
9871 call	Constant	=nullptr
9872	Constant	TYPE(ABEND_PROG)
9873 Output		
9874 t_errno	<----	RETURN_CODE(variable)

9875 **Table I-18** *t_snddis* (Incoming Connect Req.) <--> FDX Verbs and Parameters

9876 XTI Function and Parameters	9877 <--Relation-->	9878 FDX Verb & Parameter
9878 t_snddis()		DEALLOCATE
9879 To reject incoming connect request		
9881 Input		
9882 call-->sequence	Used Locally	
9883 Output		
9884 t_errno	<----	RETURN_CODE(variable)

9885 **Table I-19** *t_sndrel* <--> FDX Verbs and Parameters

9886 XTI Function and Parameters	9887 <--Relation-->	9888 FDX Verb & Parameter
9888 t_sndrel()		DEALLOCATE
9889 Input		
9890 fd	Used Locally	
9891	Created Locally	RESOURCE(variable)
9892	Created Locally	TYPE(FLUSH)
9893 Output		
9894 t_errno	<----	RETURN_CODE(variable)

9895 I.3.4 Half Duplex Mapping

9896 The interface to the SNA transport provider is the FDX LU 6.2 interface. If the SNA transport
9897 provider does not support FDX LU 6.2, the FDX verbs can be mapped to two half-duplex LU 6.2
9898 connections, one used to send in each direction. This gives the appearance of a full duplex
9899 connection without requiring any conversation direction turn-arounds on the half-duplex
9900 conversations.

9901 The mapping of FDX LU 6.2 verbs to two half-duplex connections is described in FDX Kit.

9902 **I.3.5 Return Code to Event Mapping**

9903 The following table Table I-20 shows how the return codes on LU 6.2 verbs are mapped to XTI
 9904 events.

9905 Any return code for which there no mapping given in this table will create a disconnect.

9906 **Table I-20** Mapping of XTI Events to SNA Events

XTI Event	Full Duplex SNA Event
T_CONNECT	ALLOCATE completes with RETURN_CODE=OK
T_DATA	RECEIVE_AND_WAIT completes with OK return code and WHAT_RECEIVED = <ul style="list-style-type: none"> • DATA_COMPLETE • DATA_INCOMPLETE
T_DISCONNECT	One of the following has occurred: SEND_DATA issued with RETURN_CODE = ERROR_INDICATION with subcode from list below: <ul style="list-style-type: none"> • ALLOCATION_ERROR • DEALLOCATE_ABEND_PROG • DEALLOCATE_ABEND_SVC • DEALLOCATE_ABEND_TIMER • RESOURCE_FAILURE_NO_RETRY • RESOURCE_FAILURE_RETRY Any other verb issued with RETURN_CODE of <ul style="list-style-type: none"> • ALLOCATION_ERROR • DEALLOCATE_ABEND_PROG • DEALLOCATE_ABEND_SVC • DEALLOCATE_ABEND_TIMER • RESOURCE_FAILURE_NO_RETRY • RESOURCE_FAILURE_RETRY
T_EXDATA	RECEIVE_EXPEDITED_DATA completes with OK return code
T_GODATA	Flow control restrictions on normal data flow that lead to a [TFLOW] error have been lifted. Normal data may be sent again.
T_GOEXDATA	Flow control restrictions on expedited data flow that lead to a [TFLOW] error have been lifted. Expedited data may be sent again.
T_LISTEN	When the partner program is instantiated when the connection request arrives (the typical LU 6.2 model), this event is posted as soon as the program issues the <i>t_listen()</i> function call. When the partner program already exists, this event is posted when the connection request arrives and is matched with the program.
T_ORDREL	Set when a RECEIVE_* verb completes with RETURN_CODE = DEALLOCATE_NORMAL.

9946 **I.4 XTI SNA CR**

9947 Reference is made in this Appendix (see item 7 in Section I.2.1 on page 294, and the description
 9948 for *snd()* in Section I.2.4 on page 298) to a T_PUSH flag on the *t_snd()* function. The change
 9949 request below proposes extending the functionality of the X/Open Transport Interface, to add
 9950 this T_PUSH flag. It is intended that this change proposal will be decided along with other
 9951 proposals to extend XTI functionality, in the near future.

9952 For convenience, a copy of this CR, numbered 20-02 for the purposes of this publication, is
 9953 included in this Appendix. If accepted, this CR will be removed from this Appendix when the
 9954 extended functionality it proposes is incorporated into the XTI *snd()* function description in
 9955 *t_snd()* on page 93.

9956 Document: X/Open Transport Interface (XTI), CAE Specification, Version 2

9957 Change Number: 20-02

9958 Title: T_PUSH flag needed on *t_snd()*.

9959 Qualifier: Major Technical

9960 Rationale: In the XTI Appendix I, SNA Transport Provider, reference
 9961 is made to a T_PUSH flag on the XTI *t_snd()* function.
 9962 This CR proposes change to add the T_PUSH flag to the
 9963 *t_snd()* function.

9964 Change: Add the following text to the description of the flags
 9965 on the *t_snd()* definition.
 9966 This will be the third flag on the *t_snd()* function
 9967 description in the XTI specification.

9968 T_PUSH If set in flags, the transport provider
 9969 will flush all data that is currently
 9970 in its send buffers. If not set in flags,
 9971 the transport provider is free to collect
 9972 data in a send buffer until it accumulates
 9973 a sufficient amount for transmission.

The Internet Protocols

9974

9975 The Internet Protocol (IP) family is a collection of protocols designed for use in the Internet and
9976 using the Internet address format. The Internet family provides protocol support for the
9977 **SOCK_STREAM** and **SOCK_DGRAM** socket types.

9978 Internet addresses are 4-byte quantities, stored in network byte order (on “little-endian”
9979 machines, these are word and byte reversed). The `<netinet/in.h>` header defines this address as
9980 a discriminated union.

9981 The address `INADDR_ANY` can be given in a `bind()` call on a socket that uses the TCP or UDP
9982 protocol. For TCP, this lets the socket accept connections targeted at any of the host’s IP
9983 addresses. For UDP, it lets the socket accept packets addressed to any of the host’s IP addresses.
9984 The address `INADDR_BROADCAST` is the IP broadcast address. If `INADDR_BROADCAST` is
9985 used in a `sendto()` call on a UDP socket, and the host has one or more network interfaces that
9986 support the broadcast feature, a broadcast packet will be sent via one or more of those interfaces.

9987 The Internet protocol family includes the following protocols:

- 9988 • the IP transport protocol
- 9989 • the Internet Control Message Protocol (ICMP)
- 9990 • the Transmission Control Protocol (TCP)
- 9991 • the User Datagram Protocol (UDP).

9992 TCP is used to support the `SOCK_STREAM` abstraction while UDP is used to support the
9993 `SOCK_DGRAM` abstraction.

9994 The 32-bit Internet address contains both network and host parts. It is frequency-encoded; the
9995 most-significant bit is clear in Class A addresses, in which the high-order 8 bits are the network
9996 number. Class B addresses use the high-order 16 bits as the network field, and Class C
9997 addresses have a 24-bit network part. Sites with a cluster of local networks and a connection to
9998 the Internet may choose to use a single network number for cluster; this is done by using subnet
9999 addressing. The local (host) portion of the address is further subdivided into subnet and host
10000 parts. Within a subnet, each subnet appears to be an individual network; externally, the entire
10001 cluster appears to be a single, uniform network requiring only a single routing entry.

Glossary

10002

10003

abortive release

10004

An abrupt termination of a transport connection, which may result in the loss of data.

10005

asynchronous mode

10006

The mode of execution in which transport service functions do not wait for specific asynchronous events to occur before returning control to the user, but instead return immediately if the event is not pending.

10007

10008

10009

connection establishment

10010

The phase in connection mode that enables two transport users to create a transport connection between them.

10011

10012

connection mode

10013

A mode of transfer where a logical link is established between two endpoints. Data is passed over this link by a sequenced and reliable way.

10014

10015

connectionless mode

10016

A mode of transfer where different units of data are passed through the network without any relationship between them.

10017

10018

connection release

10019

The phase in connection mode that terminates a previously established transport connection between two users.

10020

10021

datagram

10022

A unit of data transferred between two users of the connectionless-mode service.

10023

data transfer

10024

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

10025

10026

expedited data

10027

Data that are considered urgent. The specific semantics of expedited data are defined by the transport provider that provides the transport service.

10028

10029

expedited transport service data unit

10030

The amount of expedited user data, the identity of which is preserved from one end of a transport connection to the other (that is, an expedited message).

10031

10032

host byte order

10033 UX

The implementation-dependent byte order supported by the local host machine (see **network byte order**). Functions are provided to convert 16 and 32-bit values between network and host byte order (see *htonl()*).

10034

10035

10036

initiator

10037

An entity that initiates a connect request.

10038

network byte order

10039 UX

The byte order in which the most significant byte of a multibyte integer value is transmitted first. This byte order is the standard byte order for Internet protocols.

10040

10041

network host database

10042 UX

A database whose entries define the names and network addresses of host machines. See *gethostent()*.

10043

10044	network net database
10045 UX	A database whose entries define the names and network numbers of networks. See <i>getnetent()</i> .
10046	network protocol database
10047 UX	A database whose entries define the names and protocol numbers of protocols. See
10048	<i>getprotoent()</i> .
10049	network service database
10050 UX	A database whose entries define the names and local port numbers of services. See <i>getservent()</i> .
10051	orderly release
10052	A procedure for gracefully terminating a transport connection with no loss of data.
10053	responder
10054	An entity with whom an initiator wishes to establish a transport connection.
10055	socket
10056 UX	A communications endpoint associated with a file descriptor that provides communications
10057	services using a specified communications protocol.
10058	synchronous mode
10059	The mode of execution in which transport service functions wait for specific asynchronous
10060	events to occur before returning control to the user.
10061	transport address
10062	The identifier used to differentiate and locate specific transport endpoints in a network.
10063	transport connection
10064	The communication circuit that is established between two transport users in connection mode.
10065	transport endpoint
10066	The communication path, which is identified by a file descriptor, between a transport user and a
10067	specific transport provider. A transport endpoint is called passive before, and active after, a
10068	relationship is established, with a specific instance of this transport provider, identified by the
10069	TSAP.
10070	transport provider identifier
10071	A character string used by the function to identify the transport service provider.
10072	transport service access point
10073	A TSAP is a uniquely identified instance of the transport provider. A TSAP is used to identify a
10074	transport user on a certain endsystem. In connection mode, a single TSAP may have more than
10075	one connection established to one or more remote TSAPs; each individual connection then is
10076	identified by a transport endpoint at each end.
10077	transport service data unit
10078	A unit of data transferred across the transport service with boundaries and content preserved
10079	unchanged. A TSDU may be divided into sub-units passed between the user and XTI. The
10080	T_MORE flag is set in all but the last fragment of a TSDU sequence constituting a TSDU. The
10081	T_MORE flag implies nothing about how the data is handled and passed to the lower level by
10082	the transport provider, and how they are delivered to the remote user.
10083	transport service provider
10084	A transport protocol providing the service of the transport layer.
10085	transport service user
10086	An abstract representation of the totality of those entities within a single system that make use
10087	of the transport service.

Glossary

10088 **user application**
10089 The set of user programs, implemented as one or more process(es) in terms of UNIX semantics,
10090 written to realise a task, consisting of a set of user required functions.

Index

<arpa/inet.h>.....	184	cmsghdr.....	155
<fcntl.h>.....	154	CO.....	265
<netdb.h>.....	185	Common Usage C.....	3
<netinet/in.h>.....	187	compatibility	
<sys/socket.h>.....	155	future.....	52
<sys/stat.h>.....	158	compilation environment.....	4
<sys/un.h>.....	159	connect indication.....	31, 53-54, 69, 86
<unistd.h>.....	188	connect request.....	48, 69, 84, 96
<xti.h>.....	47, 69, 253	connect().....	112
_XOPEN_SOURCE.....	5	connect1.....	29, 33
abortive release.....	22, 96, 329	connect2.....	29, 33
accept.....	196	connection.....	11, 57, 82, 84, 96, 98
accept().....	107	connection establishment.....	17, 19, 57-58, 84, 199, 207, 329
accept1.....	29, 33	connection mode.....	13, 17, 19, 25, 33, 190, 205, 207-208, 329
accept2.....	29, 33	connection mode service.....	33
accept3.....	29, 33	connection release.....	17, 22, 207, 329
address.....	17-19, 23, 25, 40, 43-55, 57-60, 63, 66, 69, 73-74, 89-90, 99-100, 189	connection-oriented.....	25, 207-208
address Class.....	327	connection-oriented mode.....	17, 205
application.....	11-14, 211, 252	connectionless.....	25, 198, 210
applications.....	47	connectionless mode.....	13, 17, 25, 32, 89, 91, 99, 194, 210, 329
portability.....	211, 251	constants.....	47
applications portability.....	11, 252	control message protocol.....	327
association-related.....	194	create	
association-related options.....	35	transport endpoint.....	18
asynchronous.....	34, 71	current event.....	27, 32, 71
asynchronous events.....	14	current state.....	12, 27, 32, 68, 102
asynchronous mode.....	14, 84, 329	data.....	82, 86, 89, 93, 96, 99
bind.....	11, 25, 29, 32, 53, 55, 104	data transfer.....	17, 20, 23, 32-33, 207, 329
bind().....	109	data unit.....	21, 23, 89, 99
broadcast.....	327	discarded.....	24
buffer.....	51, 61, 82	datagram.....	13, 23, 329
byte order of Internet address.....	327	datagram structure.....	258
C language		de-initialisation.....	17-18, 23, 32
Issue 4 environment.....	3	default.....	73, 211
Call structure.....	257	device.....	211
caller.....	25, 58, 69	device driver.....	252
can.....	1	discarded data unit.....	24
character string.....	11	disconnect.....	15, 17, 34, 86
checksum check.....	202	indication.....	29
child process.....	12	request.....	96
CL.....	265	disconnect structure.....	257
Class of address.....	327	dup.....	11-12, 102-103
close.....	25, 32-33, 56	duplex.....	20
close().....	111	EBADF	
closed.....	29, 32-33	in recvmsg().....	133
cluster of local networks.....	327	EINVAL.....	51

- EM212, 265
- endhostent()**162**
- endnetent()**164**
- endprotoent()**165**
- endservent()**166**
- enqueue19, 25
- errmsg**60**
- errno**60**
- errnum101
- error254
- error code91, 101
- error codes253
 - TACCES253
 - TADDRBUSY253
 - TBADADDR253
 - TBADDATA253
 - TBADF253
 - TBADFLAG253
 - TBADNAME253
 - TBADOPT253
 - TBADQLEN253
 - TBADSEQ253
 - TBUFOVFLW253
 - TFLOW253
 - TINDOUT253
 - TLOOK253
 - TNOADDR253
 - TNODATA253
 - TNODIS253
 - TNOREL253
 - TNOSTRUCTYPE253
 - TNOTSUPPORT253
 - TNOUDERR253
 - TOUTSTATE253
 - TPROTO253
 - TPROVMISMATCH253
 - TQFULL253
 - TRESADDR253
 - TRESQLEN253
 - TSTATECHNG253
 - TSYSERR253
- error handling13
- error indication91
- error message60, 101
- error number13, 101
- error numbers7
- established
 - connection208
- ETSDU21, 73-74, 82, 93-95, 265
- event27, 32, 210
 - current27, 32, 71
- event management16
- Event Management212
- events213, 254
 - accept129, 33
 - accept229, 33
 - accept329, 33
 - bind29, 32
 - closed29, 32-33
 - connect129, 33
 - connect229, 33
 - incoming30
 - listen30, 33, 69
 - opened29, 32
 - optmgmt29, 32
 - outgoing29
 - pass_conn30, 33
 - rcv30, 33, 82
 - rcvconnect30, 33
 - rcvdis130, 33
 - rcvdis230, 33
 - rcvdis330, 33
 - rcvrel30, 33
 - rcvudata30, 32
 - rcvuderr30, 32
 - snd29, 33
 - snddis129, 33
 - snddis229, 33
 - sndrel29, 33
 - sndudata29, 32
 - T_CONNECT254
 - T_DATA254
 - T_DISCONNECT254
 - T_EXDATA254
 - T_GODATA254
 - T_GOEXDATA254
 - T_LISTEN254
 - T_ORDREL254
 - T_UDERR254
 - unbind29, 32
- events and t_look15
- example218, 230
- exceptfds229
- exec102-103
- execution mode14, 21
- expanded flag93, 190, 198-199, 217, 229, 261, 329
- expedited transport service data unit329
 - ETSDU74, 82
- family of protocols327
- fcntl11, 14, 58, 69-70, 82-84, 89-90, 93, 99-100
- fcntl()**115**
- fcntl.h73

Index

fd	11, 29	ICMP	327
features	25	implementation-dependent.....	1
fgetpos()	116	INADDR_ANY	187, 327
file descriptor	11, 56, 63, 73, 102, 217, 229	INADDR_BROADCAST	187
file.c	47	incoming events	30
flag	73, 76, 82	inet_addr()	175
flags	73, 76, 82, 198, 254, 256	INET_IP	263
T_CHECK	254	inet_lnaof()	175
T_CURRENT	254	inet_makeaddr()	175
T_DEFAULT	254	inet_netof()	175
T_EXPEDITED	254	inet_network()	175
T_FAILURE	254	inet_ntoa()	175
T_MORE	254	initialisation	17-18, 23, 32, 73
T_NEGOTIATE	254	initiator	17, 329
T_NOTSUPPORT	254	interfaces	
T_PARTSUCCESS	254	implementation	2
T_READONLY	254	use	2
T_SUCCESS	254	Internet address	
flow control	24	byte order	327
fork	11-12, 102-103	Internet Protocol family	327
fsetpos()	117	Internet protocol family	327
ftell()	118	Internet protocol-specific information	199
full duplex	20	in_addr	184, 187
F_GETOWN	115, 154	in_addr_t	184-185
F_SETOWN	115, 154	in_port_t	184-185, 187
General purpose defines	259	IP transport protocol	327
gethostbyaddr()	162 , 168	IP-level options	202
gethostbyname()	162	IP-level Options	263
gethostent()	162	IPPORT_RESERVED	185
gethostname()	169	IPPROTO_ macros	
getnetbyaddr()	164 , 170	defined in <netinet/in.h>	187
getnetbyname()	164	IP_BROADCAST	202, 263
getnetent()	164	IP_DONTROUTE	202, 263
getpeername()	119	IP_OPTIONS	202, 263
getprotobyname()	165	IP_REUSEADDR	203, 263
getprotobynumber()	165 , 171	IP_TOS	203, 263
getprotoent()	165	IP_TOS type of service	263
getservbyname()	166	IP_TTL	204, 263
getservbyport()	166 , 172	ISO	189, 260, 265
getservent()	166	priorities	260
getsockname()	120	protection levels	260
getsockopt()	121	transport classes	260
headers		ISO C	3
<xti.h>	253	ISO_TP	261
host byte order	329	language-dependent	60
host part of address	327	library functions	254
hostent	185	library structure	51
htonl()	174	linger	156
htons()	174	listen	54, 69, 196
h_errno	173	listen()	123
h_errno()	162	listener application	12

- little-endian.....327
- lseek()124
- management options.....192, 195, 262
- mandatory features.....211
- maximum size
 - address.....63, 74
 - address buffer.....53, 66
 - buffer58, 69, 84, 89, 91
 - ETSDU64, 74, 95
 - TSDU21, 63, 74, 95, 206
- may1
- memory
 - allocate.....51, 61
- mode
 - asynchronous14
 - connection-oriented19, 25, 33, 190, 205, 207-208
 - connectionless23, 25, 32, 89, 91, 99, 194, 210
 - record-oriented21
 - stream-oriented.....21
 - synchronous14, 71
- modes of service13
- msghdr.....155
- MSG_ macros
 - defined in <sys/socket.h>156
- multiple options.....41
- must1
- name space
 - X/Open.....5
- netbuf structure36, 51, 76
- netent.....185
- network byte order.....329
- network host database329
- network net database.....330
- network protocol database.....330
- network service database330
- next state.....32
- NEXTHDR259
- ntohl().....174, 177
- ntohs()174
- NULL.....53
- null
 - call.....96
- null pointer51-53, 58, 60-61, 75, 84, 86, 91
- obsolescent1
- ocnt29
- open73
- opened29, 32
- option
 - buffer36
 - value42
- option management.....259
- option negotiation
 - initiate38
 - response.....39
- option values190
- options
 - association-related35
 - connection mode190
 - connectionless mode194
 - expedited data.....190
 - format.....45
 - generalities.....35
 - illegal.....37
 - IP-level.....202
 - ISO-specific260-261
 - management.....192
 - multiple41
 - privileged41
 - quality of service190
 - read-only41
 - retrieving information.....40
 - TCP-level.....200
 - transport endpoint76
 - transport level9
 - transport provider.....58
 - UDP-level.....201
 - unsupported.....38
- Options management structure257
- optmgmt.....25, 29, 32
- orderly release22, 88, 98, 199, 330
- OSI265
 - transport classes197
- outgoing events.....29
- outstanding connect indications.....31, 54, 86
- O_NONBLOCK flag14
- pass_conn30, 33
- permissions.....207
- poll71
- poll()125, 215
- POLLIN217
- polling.....15
- POLLOUT217
- POLLPRI.....217
- POLLRDBAND.....217
- POLLRDNORM.....217
- POLLWRBAND217
- portability.....46
- portable11, 25, 211, 217, 229, 251
- precedence levels
 - IP263
- primitives.....14-15
- process12

Index

program	47
programs	
multiple protocol	211
protddp	26, 35, 53-54, 57, 63, 66, 73, 76, 91, 189, 211
protocol independence	64, 74, 211
protocol-specific servicelimits	256
protocols in Internet family	327
protoent	185
PUSH flag	199
quality of service	190, 194, 261
queue	19, 25, 70
rate	190
rate structure	260
rcv	30, 33
rcvconnect	30, 33
rcvdis1	30, 33
rcvdis2	30, 33
rcvdis3	30, 33
rcvrel	30, 33
rcvudata	30
rcvuderr	30
read()	126
readfds	229
readv()	126
reason	
disconnect	86
receipt	88
receive	82, 89
Receiving Data	20, 23
record-oriented	21
recv()	127
recvfrom()	129
recvmsg()	132
release	17, 22, 33, 88, 96, 98
reliable	13
remote user	16, 19, 22, 56, 58, 96, 208, 210
reqvalue	190, 261
reqvalue structure	261
resfd	29
responder	17, 330
sa_family_t	155
select()	135, 227
send()	136
Sending Data	21, 24
sendmsg()	138
sendto()	141
servent	185
server program	217, 229
service definition	
ISO	22, 189, 196
TCP	22
service type defines	256
sethostent()	162, 178
setnetent()	164, 179
setprotoent()	165, 180
setserverent()	166, 181
setsockopt()	144
should	1
shutdown()	146
snd	29, 33-34, 189
snddis1	29, 33
snddis2	29, 33
sndrel	29, 33-34
sndudata	29, 32, 34, 189
sockaddr	155
sockaddr_in	187
sockaddr_un	159
socket	205-206, 330
socket()	147
socketpair()	149
SOCK_DGRAM	327
SOCK_RAW	327
SOCK_STREAM	327
SO_ macros	
defined in <sys/socket.h>	156
standard error	60
state	27-28, 32, 259
current	27, 32, 68, 102
next	32
T_DATAXFER	28, 259
T_IDLE	28, 259
T_INCON	28, 259
T_INREL	28, 259
T_OUTCON	28, 259
T_OUTREL	28, 259
T_UNBIND	28
T_UNBND	259
T_UNIT	28
state table	32-33, 210
status	
connect request	20, 84
connection	58
stream-oriented	21
strerror(3C)	60
struct netbuf	256
struct rate	260
struct reqvalue	261
struct thrpt	261
struct transdel	261
struct t_bind	257
struct t_call	257
struct t_discon	257

struct t_info.....	256	TCO_NETEXP.....	262
struct t_kpalive.....	262	TCO_NETRECPTCF.....	262
struct t_linger.....	260	TCO_PRIORITY.....	261
struct t_opthdr.....	257	TCO_PROTECTION.....	261
struct t_optmgmt.....	257	TCO_REASTIME.....	262
struct t_uderr.....	258	TCO_RELDELAY.....	261
struct t_unitdata.....	258	TCO_RELFAILPROB.....	261
structure types.....	258	TCO_RESERRORRATE.....	261
T_BIND.....	258	TCO_THROUGHPUT.....	261
T_CALL.....	258	TCO_TRANSDEL.....	261
T_DIS.....	258	TCO_TRANSFFAILPROB.....	261
T_INFO.....	258	TCP.....	22, 199, 205, 265, 327
T_OPTMGMT.....	258	TCP-level options.....	200, 262
T_UDERROR.....	258	TCP_KEEPALIVE.....	200, 262
T_UNITDATA.....	258	TCP_MAXSEG.....	201, 262
subnet part of address.....	327	TCP_NODELAY.....	201, 262
SVID.....	265	terminated	
synchronise.....	102	connection.....	208
synchronous mode.....	14, 71, 330	terminology.....	1
t-opthdr.....	190	TFLOW.....	24, 213, 253
TACCES.....	253	thrt.....	190, 261
TADDRBUSY.....	253	thrt structure.....	261
TBADADDR.....	253	TINDOUT.....	253
TBADDATA.....	253	TLI.....	251-252, 265
TBADF.....	253	TLOOK.....	15, 21, 34, 253
TBADFLAG.....	253	TNOADDR.....	253
TBADNAME.....	253	TNODATA.....	253
TBADOPT.....	253	TNODIS.....	253
TBADQLEN.....	253	TNOREL.....	253
TBADSEQ.....	253	TNOSTRUCTYPE.....	253
TBUFOVFLW.....	253	TNOTSUPPORT.....	253
TC.....	265	TNOUDERR.....	253
TCL_CHECKSUM.....	262	TOS precedence levels.....	263
TCL_PRIORITY.....	261	TOUTSTATE.....	253
TCL_PROTECTION.....	261	TPDU lengths.....	260
TCL_RESERRORRATE.....	261	TPROTO.....	253
TCL_TRANSDEL.....	261	TPROVMISMATCH.....	253
TCO_ACKTIME.....	262	TQFULL.....	253
TCO_ALTCLASS1.....	262	transdel.....	190
TCO_ALTCLASS2.....	262	transdel structure.....	261
TCO_ALTCLASS3.....	262	Transmission Control Protocol.....	327
TCO_ALTCLASS4.....	262	transport address.....	11, 189, 330
TCO_CHECKSUM.....	262	transport classes.....	197, 260
TCO_CLASS.....	262	transport connection.....	11, 19, 56, 63, 98, 330
TCO_CONNRESIL.....	261	transport error codes.....	11, 19, 56, 63, 98, 330
TCO_ESTDELAY.....	261	Transport Level Interface (TLI).....	251-252
TCO_ESTFAILPROB.....	261	transport level options.....	9
TCO_EXPD.....	261	transport options.....	11, 17, 73, 330
TCO_EXTFORM.....	262	transport provider identifier.....	11, 17, 73, 330
TCO_FLOWCTRL.....	262	transport service.....	9, 189, 207
TCO_LTPDU.....	262	transport service access point.....	330

Index

TSAP.....	12	T_DISCONNECT.....	15-16, 21, 34, 213, 254
transport service data unit.....	330	t_errno.....	13, 101, 254
TSDU.....	16, 74, 82, 211	t_error.....	13, 25, 60, 101, 254
transport service provider.....	330	t_error().....	60
transport service user.....	17, 19, 22, 27-28, 57, 207, 330	T_EXDATA.....	15-16, 205, 213, 254
transport user actions.....	31	T_EXPEDITED.....	21, 82, 254
TRESADDR.....	253	T_FAILURE.....	254
TRESQLEN.....	253	T_FLASH.....	263
TSAP.....	12, 265	t_free.....	25, 61, 254
TSDU 16, 20, 73-74, 82, 93-95, 99, 189, 198-199, 265		t_free().....	61
TSTATECHNG.....	253	t_getinfo.....	25, 63, 196, 255
TSYSERR.....	13, 21, 60, 253	t_getinfo().....	63
T_ABSREQ.....	259	t_getprotaddr.....	25, 66, 254
t_accept.....	25, 48, 196, 199, 205, 254	t_getprotaddr().....	66
t_accept().....	48	t_getstate.....	25, 68, 255
T_ACTIVEPROTECT.....	260	t_getstate().....	68
T_ADDR.....	258	T_GODATA.....	15-16, 23, 213, 254
T_ALIGN.....	259	T_GOEXDATA.....	15-16, 213, 254
T_ALL.....	258	T_HIRES.....	263
t_alloc.....	25, 51, 61, 254, 258	T_HITHRPT.....	263
t_alloc().....	51	T_IDLE.....	28, 32-33, 57, 259
T_ALLOPT.....	259	T_IMMEDIATE.....	263
t_bind.....	11, 25, 53, 104, 196, 205, 254	T_INCON.....	28, 33, 259
T_BIND.....	258	T_INETCONTROL.....	263
t_bind().....	53	T_INFINITE.....	259
t_call.....	48	T_INFO.....	258
T_CALL.....	258	T_INREL.....	28, 33, 259
T_CHECK.....	254	T_INVALID.....	259
T_CLASS0.....	260	t_kpalive.....	262
T_CLASS1.....	260	T_LDELAY.....	263
T_CLASS2.....	260	T_LISTEN.....	15
T_CLASS3.....	260	t_listen.....	15
T_CLASS4.....	260	T_LISTEN.....	16
t_close.....	25, 56, 207, 254	t_listen.....	25
t_close().....	56	T_LISTEN.....	34
T_CLTS.....	256	t_listen.....	34, 69, 197, 199, 205
T_CONNECT.....	15	T_LISTEN.....	213, 254
t_connect.....	15	t_listen.....	255
T_CONNECT.....	16	t_listen().....	40, 69
t_connect.....	25, 57, 84, 196, 205	t_look.....	15, 25, 71, 205, 255
T_CONNECT.....	213, 254	t_look().....	71
t_connect.....	254	T_LTPDUDFLT.....	260
t_connect().....	40, 57	T_MORE.....	21, 82, 93, 254
T_COTS.....	256	T_MORE flag.....	199
T_COTS_ORD.....	256	T_NEGOTIATE.....	254
T_CRITIC_ECP.....	263	T_NETCONTROL.....	263
T_CURRENT.....	254	T_NO.....	259
T_DATA.....	15-16, 23, 33-34, 213, 254	T_NOPROTECT.....	260
T_DATAXFER.....	28, 33, 259	T_NOTOS.....	263
T_DEFAULT.....	254	T_NOTSUPPORT.....	254
T_DIS.....	15, 34, 258	T_NULL.....	259

- t_open11, 25, 73, 197, 205, 255
- t_open()73
- T_OPT258
- t_optmgmt25, 76, 255
- T_OPTMGMT258
- t_optmgmt()41, 76
- T_ORDREL15-16, 34, 213, 254
- T_OUTCON28, 33, 259
- T_OUTREL28, 33, 259
- T_OVERRIDEFLASH263
- T_PARTSUCCESS254
- T_PASSIVEPROTECT260
- T_PRIDFLT260
- T_PRIHIGH260
- T_PRILOW260
- T_PRIMID260
- T_PRIORITY263
- T_PRITOP260
- t_rcv15, 25, 34, 82, 198, 206, 255
- t_rcv()82
- t_rcvconnect25, 34, 84, 198, 206
- t_rcvconnect()40, 84
- t_rcvdis15, 25, 86, 198, 206, 255
- t_rcvdis()86
- t_rcvrel15, 34, 88, 199, 255
- t_rcvrel()88
- t_rcvudata25, 34, 89, 198, 255
- t_rcvudata()40, 89
- t_rcvuderr15, 25, 91, 198, 255
- t_rcvuderr()40, 91
- T_READONLY254
- T_ROUTINE263
- T_SENDZERO256
- t_snd15, 25, 93, 198, 206, 255
- t_snd()93
- t_snddis25, 96, 198, 206, 255
- t_snddis()96
- t_sndrel98, 199, 255
- t_sndrel()98
- t_sndudata25, 99, 198, 206, 255
- t_sndudata()99
- t_strerror25, 101, 255
- t_strerror()101
- T_SUCCESS254
- t_sync12, 25, 102, 255
- t_sync()102
- T_UDATA258
- T_UDERR15-16, 23, 34, 213, 254
- T_UDERROR258
- t_unbind25
- T_UNBIND28
- t_unbind104, 255
- t_unbind()104
- T_UNBND32-33, 259
- T_UNIT28
- T_UNITDATA258
- T_UNSPEC44, 259
- T_UNUSED259
- T_YES259
- UDP199, 265, 327
- UDP-level options201, 262
- UDP_CHECKSUM202, 263
- unbind25, 29, 32, 34
- undefined1
- unitdata89, 99
- Unitdata error structure258
- UNIX
 - process12
 - versions251
- unspecified1
- user application17, 23, 331
- user data86
- User Datagram Protocol327
- UX2, 214, 226, 254-255, 329-330
 - in <arpa/inet.h>184
 - in <fcntl.h>154
 - in <netdb.h>185
 - in <netinet/in.h>187
 - in <sys/socket.h>155
 - in <sys/stat.h>158
 - in <sys/un.h>159
 - in accept()107
 - in bind()109
 - in close()111
 - in connect()112
 - in endhostent()162
 - in endnetent()164
 - in endprotoent()165
 - in endservent()166
 - in fcntl()115
 - in fgetpos()116
 - in fsetpos()117
 - in ftell()118
 - in gethostbyaddr()168
 - in gethostname()169
 - in getnetbyaddr()170
 - in getpeername()119
 - in getprotobyname()171
 - in getservbyport()172
 - in getsockname()120
 - in getsockopt()121
 - in htonl()174

Index

in h_errno.....	173
in inet_addr().....	175
in listen().....	123
in lseek().....	124
in ntohl().....	177
in poll().....	125
in read().....	126
in recv().....	127
in recvfrom().....	129
in recvmsg().....	132
in select().....	135
in send().....	136
in sendmsg().....	138
in sendto().....	141
in sethostent().....	178
in setnetent().....	179
in setprotoent().....	180
in setservent().....	181
in setsockopt().....	144
in shutdown().....	146
in socket().....	147
in socketpair().....	149
in write().....	151
warning	
UX.....	2
will.....	2
write().....	151
writetds.....	229
writev().....	151
X/Open name space.....	5
XEM.....	265
XTI.....	9, 265
applications.....	47
features.....	25
library.....	47
XTI error return.....	254
XTI-level options.....	79, 259
XTI_DEBUG.....	259
XTI_GENERIC.....	259
XTI_LINGER.....	259
XTI_RCVBUF.....	259
XTI_RCVLOWAT.....	259
XTI_SNDBUF.....	259
XTI_SNTLOWAT.....	260
Zero-length TSDUs and TSDU fragments.....	189, 198

