*Technical Standard*

# Networking Services (XNS) Issue 5.2

*The Open Group*



Technical Standard

Networking Services (XNS) Issue 5.2

ISBN: 1-85912-241-8
Document Number: C808

Published in the U.K. by The Open Group, January 2000.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

# Contents


**Part 1 Part 1: Common Information ..... 1**

**Chapter 1 Common Information ..... 1**

1.1 Terminology ..... 1
1.2 Use and Implementation of Interfaces ..... 2
1.2.1 C Language Definition ..... 2
1.3 The Compilation Environment ..... 3
1.3.1 The Name Space ..... 3
1.4 Relationship to the XSH Specification ..... 6
1.4.1 Error Numbers ..... 6
1.5 Thread Safety ..... 6
1.6 Thread Cancellation Points ..... 6
1.7 Relationship to Emerging Formal Standards ..... 7

**Part 2 Part 2: Sockets ..... 9**

**Chapter 2 Sockets Interfaces ..... 11**

2.1 Sockets Overview ..... 11
*accept*( ) ..... 12
*bind*( ) ..... 14
*close*( ) ..... 16
*connect*( ) ..... 17
*fcntl*( ) ..... 20
*fgetpos*( ) ..... 21
*fsetpos*( ) ..... 22
*ftell*( ) ..... 23
*getpeername*( ) ..... 24
*getsockname*( ) ..... 25
*getsockopt*( ) ..... 26
*if_freenameindex*( ) ..... 29
*if_indextoname*( ) ..... 30
*if_nameindex*( ) ..... 31
*if_nametoindex*( ) ..... 32
*listen*( ) ..... 33
*lseek*( ) ..... 35
*poll*( ) ..... 36
*read*( ) ..... 37
*recv*( ) ..... 38
*recvfrom*( ) ..... 40
*recvmsg*( ) ..... 43
*select*( ) ..... 46

*send*( ) ..... 47
*sendmsg*( ) ..... 49
*sendto*( ) ..... 52
*setsockopt*( ) ..... 55
*shutdown*( ) ..... 58
*socket*( ) ..... 59
*socketpair*( ) ..... 61
*write*( ) ..... 63

**Chapter 3 Sockets Headers ..... 65**
**<fcntl.h>** ..... 66
**<net/if.h>** ..... 67
**<sys/socket.h>** ..... 68
**<sys/stat.h>** ..... 73
**<sys/uio.h>** ..... 74

**Chapter 4 IP Address Resolution Interfaces ..... 75**
*endhostent*( ) ..... 76
*endnetent*( ) ..... 80
*endprotoent*( ) ..... 81
*endservent*( ) ..... 82
*gai_strerror*( ) ..... 84
*getaddrinfo*( ) ..... 85
*gethostname*( ) ..... 88
*getnameinfo*( ) ..... 89
*h_errno* ..... 91
*htonl*( ) ..... 92
*inet_addr*( ) ..... 93
*inet_pton*( ) ..... 95

**Chapter 5 IP Address Resolution Headers ..... 97**
**<arpa/inet.h>** ..... 98
**<netdb.h>** ..... 99
**<netinet/tcp.h>** ..... 103
**<unistd.h>** ..... 104

**Chapter 6 Use of Sockets for Local UNIX Connections ..... 105**
**<sys/un.h>** ..... 106

**Chapter 7 Use of Sockets over Internet Protocols based on IPv4 ..... 107**
**<netinet/in.h>** ..... 108

**Chapter 8 Use of Sockets over Internet Protocols based on IPv6 ..... 111**
8.1 Addressing ..... 111
8.2 Compatibility with IPv4 ..... 112
8.3 Interface Identification ..... 112
8.4 Options ..... 113
8.5 Headers ..... 114

**<netinet/in.h> for IPv6** ..... 115

**Part 3 Part 3: XTI** ..... **117**

**Chapter 9 General Introduction to the XTI** ..... **119**

**Chapter 10 Explanatory Notes for XTI** ..... **121**
10.1 Transport Endpoints ..... 121
10.2 Transport Providers ..... 121
10.3 Association of a UNIX Process to an Endpoint ..... 122
10.4 Use of the Same Protocol Address ..... 122
10.5 Modes of Service ..... 123
10.6 Error Handling ..... 123
10.7 Synchronous and Asynchronous Execution Modes ..... 124
10.8 Effect of Signals ..... 126
10.9 Event Management ..... 126

**Chapter 11 XTI Overview** ..... **129**
11.1 Overview of Connection-oriented Mode ..... 129
11.1.1 Initialisation / De-initialisation Phase ..... 130
11.1.2 Overview of Connection Establishment ..... 131
11.1.3 Overview of Data Transfer ..... 132
11.1.4 Overview of Connection Release ..... 134
11.2 Overview of Connectionless Mode ..... 136
11.2.1 Initialisation / De-initialisation Phase ..... 136
11.2.2 Overview of Data Transfer ..... 136
11.3 XTI Features ..... 138
11.3.1 XTI Functions versus Protocols ..... 139

**Chapter 12 States and Events in XTI** ..... **141**
12.1 Transport Interfaces States ..... 142
12.2 Outgoing Events ..... 143
12.3 Incoming Events ..... 144
12.4 Transport User Actions ..... 145
12.5 State Tables ..... 146
12.6 Events and TLOOK Error Indication ..... 148

**Chapter 13 The Use of Options in XTI** ..... **149**
13.1 Generalities ..... 149
13.2 The Format of Options ..... 150
13.3 The Elements of Negotiation ..... 151
13.3.1 Multiple Options and Options Levels ..... 151
13.3.2 Illegal Options ..... 151
13.3.3 Initiating an Option Negotiation ..... 152
13.3.4 Responding to a Negotiation Proposal ..... 153
13.3.5 Retrieving Information about Options ..... 154
13.3.6 Privileged and Read-only Options ..... 155
13.4 Option Management of a Transport Endpoint ..... 156

13.5 Supplements ..... 158
13.5.1 The Option Value T_UNSPEC ..... 158
13.5.2 The info Argument ..... 158
13.5.3 Summary ..... 158
13.6 Portability Aspects ..... 160

**Chapter 14 XTI Library Functions and Parameters ..... 161**
14.1 How to Prepare XTI Applications ..... 161
14.2 Key for Parameter Arrays ..... 161
14.3 Return of TLOOK Error ..... 162
14.4 Use of ‘‘struct netbuf’’ ..... 163
*t_accept*( ) ..... 164
*t_alloc*( ) ..... 167
*t_bind*( ) ..... 169
*t_close*( ) ..... 172
*t_connect*( ) ..... 174
*t_error*( ) ..... 177
*t_errno* ..... 179
*t_free*( ) ..... 180
*t_getinfo*( ) ..... 182
*t_getprotaddr*( ) ..... 185
*t_getstate*( ) ..... 187
*t_listen*( ) ..... 188
*t_look*( ) ..... 190
*t_open*( ) ..... 192
*t_optmgmt*( ) ..... 195
*t_rcv*( ) ..... 202
*t_rcvconnect*( ) ..... 204
*t_rcvdis*( ) ..... 206
*t_rcvrel*( ) ..... 208
*t_rcvreldata*( ) ..... 209
*t_rcvudata*( ) ..... 211
*t_rcvuderr*( ) ..... 213
*t_rcvv*( ) ..... 215
*t_rcvvudata*( ) ..... 217
*t_snd*( ) ..... 219
*t_snddis*( ) ..... 222
*t_sndrel*( ) ..... 224
*t_sndreldata*( ) ..... 225
*t_sndudata*( ) ..... 227
*t_sndv*( ) ..... 229
*t_sndvudata*( ) ..... 232
*t_strerror*( ) ..... 235
*t_sync*( ) ..... 236
*t_sysconf*( ) ..... 238
*t_unbind*( ) ..... 239

**Chapter 15 The <xti.h> Header ..... 241**
**<xti.h> ..... 242**

**Chapter 16 Use of XTI with Internet Protocols ..... 251**
16.1 Introduction ..... 251
16.2 Protocol Features ..... 251
16.3 Options ..... 252
16.3.1 TCP-level Options ..... 252
16.3.2 T_UDP-level Options ..... 253
16.3.3 T_IP-level Options ..... 254
16.4 Functions ..... 257
16.5 The <xti_inet.h> Header File ..... 260
**<xti_inet.h> ..... 261**

**Part 4 Part 4: Appendixes ..... 263**

**Appendix A Use of XTI with ISO Transport Protocols ..... 265**
A.1 General ..... 265
A.2 Options ..... 266
A.2.1 Connection-mode Service ..... 266
A.2.1.1 Options for Quality of Service and Expedited Data ..... 266
A.2.1.2 Management Options ..... 268
A.2.2 Connectionless-mode Service ..... 270
A.2.2.1 Options for Quality of Service ..... 270
A.2.2.2 Management Options ..... 271
A.3 Functions ..... 272
A.4 The <xti_osi.h> Header File ..... 275
**<xti_osi.h> ..... 276**

**Appendix B Guidelines for Use of XTI ..... 279**
B.1 Transport Service Interface Sequence of Functions ..... 279
B.2 Example in Connection-oriented Mode ..... 280
B.3 Example in Connectionless Mode ..... 282
B.4 Writing Protocol-independent Software ..... 283
B.5 Event Management ..... 284
B.5.1 Short-term Solution ..... 284
B.5.2 XTI Events ..... 285
B.6 The Poll Function ..... 286
B.6.1 Example of Use of Poll ..... 286
B.7 The Select Function ..... 295
B.7.1 Example of Use of Select ..... 295

**Appendix C Use of XTI to Access NetBIOS ..... 305**
C.1 Introduction ..... 305
C.2 Objectives ..... 305
C.3 Scope ..... 306
C.4 Issues ..... 307
C.5 NetBIOS Names and Addresses ..... 307

C.6 NetBIOS Connection Release ..... 308
C.7 Options ..... 309
C.8 XTI Functions ..... 309
C.9 Compatibility ..... 313

**Appendix D XTI and TLI ..... 315**
D.1 Restrictions Concerning the Use of XTI ..... 315
D.2 Relationship between XTI and TLI ..... 316

**Appendix E Example XTI Header Files ..... 317**
E.1 Example <xti.h> Header ..... 317
E.2 Example <xti_osi.h> Header File ..... 325
E.3 Example <xti_inet.h> Header File ..... 328

**Appendix F Minimum OSI Functionality ..... 331**
F.1 General ..... 331
F.1.1 Rationale for using XTI-mOSI ..... 331
F.1.2 Migrant Applications ..... 331
F.1.3 OSI Functionality ..... 331
F.1.4 mOSI API versus XAP ..... 332
F.1.5 Upper Layers Functionality Exposed via mOSI ..... 332
F.1.5.1 Naming and Addressing Information used by mOSI ..... 332
F.1.5.2 XTI Options Specific to mOSI ..... 332
F.2 Options ..... 334
F.2.1 ACSE / Presentation Connection-mode Service ..... 334
F.2.2 ACSE / Presentation Connectionless-mode Service ..... 335
F.2.3 Transport Service Options ..... 336
F.3 Functions ..... 337
F.4 Implementors) Notes ..... 341
F.4.1 Upper Layers FUs, Versions and Protocol Mechanisms ..... 341
F.4.2 Mandatory and Optional Parameters ..... 341
F.4.3 Mapping XTI Functions to ACSE / Presentation Services ..... 342
F.4.3.1 Connection-mode Services ..... 342
F.4.3.2 Connectionless-mode Services ..... 346
F.5 Option Data Types and Structures ..... 347
F.6 <xti_mosi.h> Header File ..... 351

**Appendix G SNA Transport Provider ..... 353**
G.1 Introduction ..... 353
G.2 SNA Transport Protocol Information ..... 354
G.2.1 General ..... 354
G.2.2 SNA Addresses ..... 355
G.2.3 Options ..... 356
G.2.3.1 Connection-Mode Service Options ..... 356
G.2.4 Functions ..... 358
G.3 Mapping XTI to SNA Transport Provider ..... 361
G.3.1 General Guidelines ..... 362
G.3.2 Flows Illustrating Full Duplex Mapping ..... 363

G.3.3 Full Duplex Mapping ... 372
G.3.3.1 Parameter Mappings ... 374
G.3.4 Half Duplex Mapping ... 384
G.3.5 Return Code to Event Mapping ... 385
G.4 Compatibility ... 386

**Appendix H IPX/SPX Transport Provider ... 387**
H.1 General ... 387
H.2 Namespace ... 387
H.2.1 IPX ... 387
H.2.2 SPX ... 388
H.3 Options ... 388
H.3.1 IPX-level Options ... 388
H.3.2 SPX-level Options ... 389
H.4 Functions ... 391

**Appendix I ATM Transport Protocol Information for XTI ... 395**
I.1 General ... 395
I.2 ATM Addresses ... 396
I.2.1 ATM Network Address ... 396
I.2.2 ATM Protocol Address ... 396
I.2.3 t_atm_sap Structure ... 397
I.3 Options ... 401
I.3.1 Signalling-level Options ... 401
I.3.2 Absolute Requirements ... 402
I.3.3 Further Remarks ... 402
I.4 Existing Functions ... 413
I.5 Implementation Notes ... 416
I.6 New Functions ... 419
*t_addleaf*( ) ... 420
*t_removeleaf*( ) ... 422
*t_rcvleafchange*( ) ... 424

**Appendix J ATM Transport Protocol Information for Sockets ... 427**
J.1 General ... 427
J.2 Existing Functions ... 428
J.3 Point-to-Multipoint Connections ... 431
J.3.1 Adding a Leaf ... 431
J.3.2 Removing a Leaf ... 432
J.3.3 Receiving Indication of a Change in Leaf Status ... 432
J.4 Implementation Notes ... 434

**Appendix K ATM Transport Headers ... 437**
K.1 Proposed Additions to <xti.h> ... 437
K.2 Proposed Additions to <sys/socket.h> ... 437
K.3 <xti_atm.h> ... 437
K.4 <netatm/atm.h> ... 438
K.5 <_atm_common.h> ... 439

**Glossary** ..... **447**

**Index** ..... **451**

## List of Figures

B-1 Sequence of Transport Functions in Connection-oriented Mode ..... 281
B-2 Sequence of Transport Functions in Connectionless Mode ..... 282
G-1 Active Connection Establishment, Blocking Version (1 of 2) ..... 363
G-2 Active Connection Establishment, Non-blocking Version (2 of 2) ..... 364
G-3 Passive Connection Establishment, Instantiation Version (1 of 3) ..... 365
G-4 Passive Connection Establishment, Blocking Version (2 of 3) ..... 366
G-5 Passive Connection Establishment, Non-blocking Version (3 of 3) ..... 367
G-6 XTI Function to LU 6.2 Verb Mapping: Blocking t_snd ..... 368
G-7 XTI Function to LU 6.2 Verb Mapping: Non-blocking t_snd ..... 369
G-8 XTI Function to LU 6.2 Verb Mapping: Blocking t_rcv ..... 370
G-9 Mapping from XTI Calls to LU 6.2 Verbs (Passive side) ..... 371

## List of Tables

10-1 Events and *t_look*( ) ..... 125
11-1 Classification of the XTI Functions ..... 139
12-1 Transport Interface States ..... 142
12-2 Transport Interface Outgoing Events ..... 143
12-3 Transport Interface Incoming Events ..... 144
12-4 Transport Interface User Actions ..... 145
12-5 Initialisation / De-initialisation States ..... 146
12-6 Data Transfer States: Connectionless-mode ..... 146
12-7 Connection / Release / Data Transfer States: Connection-mode ..... 147
14-1 XTI-level Options ..... 199
16-1 TCP-level Options ..... 252
16-2 T_UDP-level Option ..... 253
16-3 T_IP-level Options ..... 254
A-1 Options for Quality of Service and Expedited Data ..... 266
A-2 Management Options ..... 268
A-3 Options for Quality of Service ..... 270
A-4 Management Option ..... 271
F-1 APCO-level Options ..... 334
F-2 APCL-level Options ..... 336
F-3 Association Establishment ..... 343
F-4 Data Transfer ..... 344
F-5 Association Release ..... 345
F-6 Connectionless-mode ACSE Service ..... 346
G-1 SNA Options ..... 357
G-2 Fields for *info* Parameter ..... 358
G-3 Default Characteristics returned by *t_open*( ) ..... 359
G-4 FDX LU 6.2 Verb Definitions ..... 362
G-5 XTI Mapping to LU 6.2 Full Duplex Verbs ..... 372

| | | |
|---|---|---|
| G-6 | Relation Symbol Description | 374 |
| G-7 | *t_accept* ↔ FDX Verbs and Parameters | 374 |
| G-8 | *t_bind* ↔ FDX Verbs and Parameters | 375 |
| G-9 | *t_close* ↔ FDX Verbs and Parameters | 375 |
| G-10 | *t_connect* ↔ FDX Verbs and Parameters | 376 |
| G-11 | *t_getprocaddr* ↔ FDX Verbs and Parameters | 378 |
| G-12 | *t_listen* ↔ FDX Verbs and Parameters | 379 |
| G-13 | *t_optmgmt* ↔ FDX Verbs and Parameters | 379 |
| G-14 | *t_rcv* ↔ FDX Verbs and Parameters | 380 |
| G-15 | *t_rcvconnect* ↔ FDX Verbs and Parameters | 381 |
| G-16 | *t_snd* ↔ FDX Verbs and Parameters | 382 |
| G-17 | *t_snddis* (Existing Connection) ↔ FDX Verbs and Parameters | 383 |
| G-18 | *t_snddis* (Incoming Connect Req.) ↔ FDX Verbs and Parameters | 383 |
| G-19 | *t_sndrel* ↔ FDX Verbs and Parameters | 383 |
| G-20 | Mapping of XTI Events to SNA Events | 385 |
| I-1 | Signaling-level Options | 401 |

# Preface

## The Open Group

The Open Group is a vendor and technology-neutral consortium which ensures that multi-vendor information technology matches the demands and needs of customers. It develops and deploys frameworks, policies, best practices, standards, and conformance programs to pursue its vision: the concept of making all technology as open and accessible as using a telephone.

The mission of The Open Group is to deliver assurance of conformance to open systems standards through the testing and certification of suppliers' products.

The Open group is committed to delivering greater business efficiency and lowering the cost and risks associated with integrating new technology across the enterprise by bringing together buyers and suppliers of information systems.

Membership of The Open Group is distributed across the world, and it includes some of the world's largest IT buyers and vendors representing both government and commercial enterprises.

More information is available on The Open Group Web Site at *http://www.opengroup.org*.

## Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available on The Open Group Web Site at *http://www.opengroup.org/pubs*.

- Product Standards

  A Product Standard is the name used by The Open Group for the documentation that records the precise conformance requirements (and other information) that a supplier's product must satisfy. Product Standards, published separately, refer to one or more Technical Standards.

  The ''X'' Device is used by suppliers to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the Open Brand Trademark License Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the supplier. The Open Group runs similar conformance schemes involving different trademarks and license agreements for other bodies.

- Technical Standards (formerly CAE Specifications)

  Open Group Technical Standards, along with standards from the formal standards bodies and other consortia, form the basis for our Product Standards (see above). The Technical Standards are intended to be used widely within the industry for product development and procurement purposes.

  Technical Standards are published as soon as they are developed, so enabling suppliers to proceed with development of conformant products without delay.

  Anyone developing products that implement a Technical Standard can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand.

- CAE Specifications

  CAE Specifications and Developers' Specifications published prior to January 1998 have the same status as Technical Standards (see above).

- Preliminary Specifications

  Preliminary Specifications have usually addressed an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. There is a strong preference to develop or adopt more stable specifications as Technical Standards.

- Consortium and Technology Specifications

  The Open Group has published specifications on behalf of industry consortia. For example, it published the NMF SPIRIT procurement specifications on behalf of the Network Management Forum (now TMF). It also published Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

In addition, The Open Group publishes Product Documentation. This includes product documentation—programmer's guides, user manuals, and so on—relating to the DCE, Motif, and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

### Versions and Issues of Specifications

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

### Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on The Open Group Web Site at *http://www.opengroup.org/corrigenda*.

### Ordering Information

Full catalog and ordering information on all Open Group publications is available on The Open Group Web Site at *http://www.opengroup.org/pubs*.

### This Document

This Networking Services (XNS) Issue 5.2 Technical Standard describes the industry-standard Open Systems interfaces to communications services. These include two APIs to transport-level process-to-process communications: Sockets, and X/Open Transport Interface (XTI).

Sockets (Part 2 of this document) is mandatory. XTI (Part 3) is optional. The XTI interface is now considered to be obsolete. Writers of new applications using the Internet protocol suite are recommended to use sockets rather than XTI. Where protocols for which there is no sockets support are in use, XTI is still recommended in preference to proprietary APIs.

Both Sockets and XTI are specified for use over Internet protocols (TCP, UDP and IP) and ISO Transport protocols. They also include a set of Internet address resolution interfaces which are commonly used in conjunction with Sockets. XTI support for many other protocols is described in appendices to XNS. Sockets and Address Resolution must be supported over the Internet protocols. XTI may be supported over either the Internet or ISO Transport protocols. Other protocols may also be provided, but this is not required for the Brand.

Branded UNIX98 systems support the Sockets, XTI and Address Resolution interfaces described in XNS Issue 5.

XNS Issue 5.2 contains a number of new features over the previous publication[1]. The most important new feature in XNS Issue 5.2 is the inclusion of Internet Protocol version 6 (IPv6) functionality, in a manner which is aligned with the relevant IETF IPv6 standard (RFC 2553). Other new features include:

- Any text that relates to behaviour of the implementation when _XOPEN_SOURCE is less than 500 is informative, not normative. This behaviour is specified normatively in earlier issues of XNS.
- Conformant systems are not required to provide the OPT_NEXTHDR macro.
- Protocol-specific symbols defined in **<xti_inet.h>** or **<xti_osi.h>** are not required to be available when **<xti.h>** is included by the application but **<xti_inet.h>** or **<xti_osi.h>** (respectively) is not included by the application.
- An implementation is only required to provide protocol-specific headers for those protocols that it supports.
- An implementation need not make available symbols marked in XNS Issue 5 as "LEGACY".
- Although identifiers marked as "LEGACY" are not specified as being reserved for any use by the implementation, implementations may make them available.

**Structure**

- Part 1 gives a common introduction. It contains information comparable to that in the **XSH** specification. It applies to the Sockets and Address Resolution interfaces (see Part2 below) if the UNIX compilation environment is in effect.
- Part 2 defines the **Sockets** interface.
- Part 3 defines the **XTI** interface.
- Part 4 includes API mapping information to other transport providers, and other guidance for implementers.

1. The previous XNS publication was Networking Services (XNS), Issue 5, CAE Specification, February 1997, (ISBN: 1-85912-165-9, C523).

   The preceeding XNS publication was Networking Services, Issue 4, CAE Specification, September 1994, (ISBN: 1-85912-049-0, C438).

### 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 [ABCD] is used to identify a return value ABCD, including if this is an 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.
- All text marked by the side-headings **Note** or **Notes** is for added information, and is non-normative.

### Change History

- **Issue 5.1 Draft 2.0:**
  Re-structured to accommodate IPv4/IPv6, but no IPv6 content added
  Corrigendum U031 inserted.
  Change Requests HP-XTI-1/2/3 applied.
- **Issue 5.1 Draft 3.0:**
  IPv6 CRs TOG:XNS5-01R2, TOG:XNS5-02R2, TOG:XNS5-03R2, TOG:XNS5-05 and TOG:XNS5-06 (ogtgnet 6865 attachments) applied, and the resulting text changes marked with a special "+" change mark (i.e. ".mc +").
  The following CRs approved at XNET53 applied, and the resulting text changes marked with the normal "|" change mark.

| | |
|---|---|
| SUN:XNS-101 | ogtgnet 6741 |
| SUN:XNS-102R1 | ogtgnet 6879 |
| SUN:XNS-103 | ogtgnet 6741 |
| SUN:XNS-001 | ogtgnet 6799 |
| SUN:XNS-002 | ogtgnet 6799 |
| SUN:XNS-003 | ogtgnet 6799 |
| SUN:XNS-005 | ogtgnet 6800 |
| SUN:XNS-006R | ogtgnet 6919 |
| TOG:XNS5-05 | ogtgnet 6761, with modification defined in the XNET-53 meeting minutes. (Note that this is a different CR from TOG:XNS5-05 in ogtgnet 6865 although it does duplicate the CR number) |

- **Issue 5.2 Draft 2.0**:
  The following Change Requests as approved in June 1999 were applied:

  TOG:XNS-001
  TOG:XNS-002
  TOG:XNS-003
  TOG:XNS-004
  TOG:XNS-005
  TOG:XNS-006
  TOG:XNS-007
  TOG:XNS-008
  TOG:XNS-009
  TOG:XNS-010
  TOG:XNS-011
  TOG:XNS-012
  TOG:XNS-013
  TOG:XNS-014

- **Issue 5.2 Draft 3.0**:
  The following Change Requests as approved in August 1999 were applied:

  SUN:XNS-001
  SUN:XNS-002
  TOG:XNS-001, with *Additional IP Address Resolution Functions* revised by:
  NRL:XNS_GAI-006,-009,-013,-014,-016,-017,-021,-022,-025.

  Further Change Requests as approved in November 1999 were applied:
  HP:XNS-001, SUN:XNS-001 - 013,015,017,018,020,021,023 - 030, TOG:XNS-001 - 008,010,011.

- **Issue 5.2 Draft 4.0**:
  This was the sanity-check copy. No sanity-check comments were received. Draft 4.0 was therefore published electonically as XNS Issue 5.2, C808.

# Trade Marks

AT&T® is a registered trademark of AT&T in the U.S.A. and other countries.

Hewlett-Packard®, HP®, HP-UX®, and Openview® are registered trademarks of Hewlett-Packard Company.

Motif®, OSF/1®, UNIX®, and the ‘‘X Device’’ are registered trademarks and IT DialTone™ and The Open Group™ are trademarks of The Open Group in the U.S. and other countries.

SNA is a product of International Business Machines Corporation.

/usr/group® is a registered trademark of UniForum, the International Network of UNIX System Users.

# 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 material.
- 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.
- The Open Group gratefully acknowledges the valued contribution of the following people in the development of this specification, along with their corporate affiliation at the time of their contribution:

| | | | |
|---|---|---|---|
| Josee Auber | Hewlett-Packard | Lori Mickelson | Unisys |
| Kathryn Britton | IBM | Laura Micks | IBM |
| Philippe Camus | Groupe Bull | Finnbarr Murphy | DIGITAL |
| Andrew Chandler | ICL | Alagu Periyannan | Apple |
| Andre Cohen | Groupe Bull | George Preoteasa | Hewlett-Packard |
| Paul Comstock | Hewlett-Packard | John Ronciak | Unisys |
| Andrew Gollan | Sun | Seth Rosenthal | Novell |
| Michel Habert | Groupe Bull | Eric Scoredos | Hewlett-Packard |
| Torez Hiley | USL | Maria Stanley | DIGITAL |
| Martin Jess | NCR | Lutz Temme | SNI |
| Mukesh Kacker | Sun | Roger Turner | IBM |
| Gerhard Kieselmann | SNI | Keith Weir | ICL |
| David Laight | Fujitsu/ICL | Robert Weirick | Unisys |
| Jack McCann | DIGITAL | Greg Wiley | SFC (SCO) |
| Hiroshi Maruta | Hitachi | Isaac Wong | Hewlett-Packard |

# Referenced Documents

The following documents are referenced in this technical standard:

**Internet**

TCP
Transmission Control Protocol, RFC 793.

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.

IPV4
Internet Protocol, RFC 791

ICMP
Internet Control Message Protocol, RFC 792

TP_ON_TCP
ISO Transport Service on Top of the TCP, RFC 1006

IPV6
Internet Protocol, Version 6, RFC 2460

IPV6_AD
IP Version 6 Addressing Architecture, RFC 2373

IPV6_BASIC_API
IPv6 Socket Interface Extensions, RFC 2553

HOSTS
DOD Internet Host Table Specification, RFC 952

**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.

**ATM**

ATMNAS ATM Forum: ''Native ATM services: Semantic Description, Version 1'', obtainable via anonymous ftp from Internet address ftp.atmforum.com, in directory /pub/approved-specs, files af-saa-0048.000.doc (Word 6.0) or af-saa-0048.000.ps (postscript).

UNIATM Forum: ''ATM User-Network Interface (UNI) Specification, Version 3.1'', published by Prentice Hall. Also obtainable electronically via anonymous ftp from Internet address ftp.atmforum.com, in directory /pub/UNI/ver3.1.

**Other References**

ISO C
ISO / IEC 9899: 1990: Programming Languages — C, including Amendment 1: 1995 (E), C Integrity (Multibyte Support Extensions (MSE) for ISO C).

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 |

Minimal OSI

ISO/IEC DISP 11188-3, International Standardized Profile — Common Upper Layer Requirements — Part 3: Minimal OSI Upper Layers Facilities, Version 6, 1994-04-14.

SVID

Networking Services Extension, System V Interface Definition (SVID) Issue 2, Volume III, 1986, UNIX Press, Morristown, NJ, USA.

NetBIOS

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

SNA

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

P1003_1G

Information Technology - Portable Operating System Interface (POSIX) - Part xx: Protocol Independent Interfaces (PII) Draft 6.6, IEEE P1003.1g/D6.6, March 1997.

RFC 1034

Domain Names - Concepts and Facilities, P. Mockapetris, November 1987.

RFC 1035

Domain Names - Implementation and Specification, P. Mockapetris, November 1987.

RFC 1886

DNS Extensions to support IP version 6, S. Thompson, C. Huitema, December 1995.

XSH, Issue 5

CAE Specification, January 1997, System Interfaces and Headers, Issue 5 (ISBN: 1-85912-181-0, C606), published by The Open Group.

XCU, Issue 5

CAE Specification, January 1997, Commands and Utilities, Issue 5 (ISBN: 1-85912-191-8, C604), published by The Open Group.

XBD, Issue 5

CAE Specification, January 1997, System Interface Definitions, Issue 5 (ISBN: 1-85912-186-1, C605), published by The Open Group.

*Technical Standard*

# Networking Services (XNS) Issue 5.2

# Part 1: Common Information

*The Open Group*

Chapter 1

# Common Information

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

## 1.1 Terminology

The information in this section applies only to the Sockets and IP Address Resolution interfaces, which are defined in Part 2.

The following terms are used in this document:

**can**
Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.

**implementation-dependent**
(Same meaning as *implementation-defined*.) Describes a value or behavior that is not defined by this document but is selected by an implementor. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations.

The implementor shall document such a value or behavior so that it can be used correctly by an application.

**legacy**
Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality.

**may**
Describes a feature or behavior that is optional for an implementation that conforms to this document. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

To avoid ambiguity, the opposite of *may* is expressed as *need not*, instead of *may not*.

**must**
Describes a feature or behavior that is mandatory for an application or user. An implementation that conforms to this document shall support this feature or behavior.

**shall**
Describes a feature or behavior that is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.

**should**
For an implementation that conforms to this document, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

**undefined**
Describes the nature of a value or behavior not defined by this document which results from use of an invalid program construct or invalid data input.

The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

**unspecified**
Describes the nature of a value or behavior not specified by this document which results from use of a valid program construct or valid data input.

The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

**will**
Same meaning as *shall*; *shall* is the preferred term.

## 1.2 Use and Implementation of Interfaces

Each of the following statements applies unless explicitly stated otherwise in the ensuing descriptions. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behaviour is undefined.

For backward compatibility purposes, any of the function names within the header file may be redefined by the implementation, using the pre-processor ‘‘#define’’ mechanism (or any other similar compiler supported mechanism) to another function name. If redefined, the function name so defined will be in accordance with the name space rules in Section 1.3.1 on page 3.

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

### 1.2.1 C Language Definition

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

## 1.3 The Compilation Environment

Applications should ensure that the feature test macro _XOPEN_SOURCE is defined with the value 520 before inclusion of any header. This is needed to enable the functionality described in this document, and possibly to enable functionality defined elsewhere in the Common Applications Environment.

The _XOPEN_SOURCE macro may be defined automatically by the compilation process, but to ensure maximum portability, applications should make sure that _XOPEN_SOURCE is defined by using either compiler options or **#define** directives in the source files, before any **#include** directives. Identifiers in this document may be undefined using the **#undef** directive as described in Section 1.3.1. These **#undef** directives must follow all **#include** directives of any headers defined in the referenced XCU specification.

Since this specification is aligned with the ISO C standard, and since all functionality enabled by _POSIX_C_SOURCE set greater than zero and less than or equal to 199506L should be enabled by _XOPEN_SOURCE set greater than or equal to 500, there should be no need to define either _POSIX_SOURCE or _POSIX_C_SOURCE if _XOPEN_SOURCE is defined. Therefore if _XOPEN_SOURCE is set greater than or equal to 500 and _POSIX_SOURCE is defined, or _POSIX_C_SOURCE is set greater than zero and less than or equal to 199506L, the behavior is the same as if only _XOPEN_SOURCE is defined and set greater than or equal to 500. However, should _POSIX_C_SOURCE be set to a value greater than 199506L, the behaviour is undefined.

The *c89* and *cc* utilities defined in the referenced XCU specification recognise the additional **–l** operand for standard libraries:

**–l xnet** If the implementation defines _XOPEN_UNIX, this operand makes visible all functions referenced in this document. An implementation may search this library in the absence of this operand.

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

If the implementation supports the utilities marked **DEVELOPMENT** in the **XCU** specification, the *lint* utility recognises the additional **–l** operand for standard libraries:

**–l xnet** Names the library **llib–lxnet.ln**, which will contain functions specified in this document.

It is unspecified whether the library **llib–lxnet.ln** exists as a regular file.

### 1.3.1 The Name Space

All identifiers in this document are defined in at least one of the headers, as shown in Chapter 3, Chapter 5 of XNS and Chapter 4 of XSH (see the referenced document **XSH**). When _XOPEN_SOURCE is defined, each header defines or declares some identifiers, potentially conflicting with identifiers used by the application. The set of identifiers visible to the application consists of precisely those identifiers from the header pages of the included headers, as well as additional identifiers reserved for the implementation. In addition, some headers may make visible identifiers from other headers as indicated on the relevant header pages.

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

1. Each identifier with external linkage described in the header section is reserved for use as an identifier with external linkage if the header is included.
2. Each macro name described in the header section is reserved for any use if the header is included.

3. Each identifier with file scope described in the header section is reserved for use as an identifier with file scope in the same name space if the header is included.

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

| Header | Prefix | Suffix |
|---|---|---|
| **<arpa/inet.h>** | in_, inet_ | |
| **<netdb.h>** | h_, n_, p_, s_ | |
| **<net/if.h>** | if_ | |
| **<netinet/in.h>** | in_, ip_, s_, sin_ | |
| **<sys/socket.h>** | _ss, sa_, if_, ifc_, ifru_, infu_, ifra_, msg_, cmsg_, l_ | |
| **<sys/un.h>** | sun_ | |
| **ANY header** | | _t |

When the optional XTI is supported, identifiers with the following prefixes are reserved for any use by the implementation:

| Header | Prefix |
|---|---|
| **<xti.h>** | l_, t_, T_ |

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

| Header | Prefix |
|---|---|
| **<net/if.h>** | IF_ |
| **<netinet/in.h>** | IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_ |
| **<netinet/tcp.h>** | TCP_ |
| **<sys/socket.h>** | AF_, CMSG_, MSG_, PF_, SCM_, SHUT_, SO |

When the optional XTI is supported, identifiers with the following prefixes are reserved for any use by the implementation:

| Header | Prefix |
|---|---|
| **<xti.h>** | OPT_, T_[2], XTI_ |

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

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

2. The following T_ prefixes are currently used: T_INET_, T_IP_, T_ISO_, T_, T_TCL_, T_TCP_, T_TCO_, T_UDP_.

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

| **Sockets:** | | | | |
|---|---|---|---|---|
| accept | getsockopt | listen | sendmsg | socketpair |
| bind | if_freenameindex | recv | sendto | |
| connect | if_indextoname | recvfrom | setsockopt | |
| getpeername | if_nameindex | recvmsg | shutdown | |
| getsockname | if_nametoindex | send | socket | |
| **IP Address Resolution:** | | | | |
| endhostent | gethostname | getprotobynumber | inet_addr | sethostent |
| endnetent | getipnodebyaddr | getprotoent | inet_lnaof | setnetent |
| endprotoent | getipnodebyname | getservbyname | inet_makeaddr | setprotoent |
| endservent | getnameinfo | getservbyport | inet_netof | setservent |
| getaddrinfo | getnetbyaddr | getservent | inet_network | |
| gethostbyaddr | getnetbyname | h_errno | inet_ntoa | |
| gethostbyname | getnetent | htonl | ntohl | |
| gethostent | getprotobyname | htons | ntohs | |

| **XTI - the following symbols are only reserved when optional XTI is supported:** | | | | |
|---|---|---|---|---|
| t_accept | t_getinfo | t_rcv | t_rcvv | t_sndudata |
| t_alloc | t_getprotaddr | t_rcvconnect | t_rcvvudata | t_sndv |
| t_bind | t_getstate | t_rcvdis | t_snd | t_sndvudata |
| t_close | t_listen | t_rcvrel | t_snd | t_strerror |
| t_connect | t_look | t_rcvreldata | t_snddis | t_sync |
| t_errno | t_open | t_rcvudata | t_sndrel | t_sysconf |
| t_error | t_optmgmt | t_rcvuderr | t_sndreldata | t_unbind |
| t_free | | | | |

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

No other identifiers are reserved.

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

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

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

## 1.4 Relationship to the XSH Specification

### 1.4.1 Error Numbers

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

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

## 1.5 Thread Safety

All interfaces defined by this document will be thread-safe, except for the following interfaces which need not be thread-safe:

*gethostbyaddr*( )
*gethostbyname*( )
*gethostent*( )
*getnetbyaddr*( )
*getnetbyname*( )
*getnetent*( )
*getprotobynumber*( )
*getprotobyname*( )
*getprotoent*( )
*getservbyname*( )
*getservbyport*( )
*getservent*( )
*inet_ntoa*( )

## 1.6 Thread Cancellation Points

Cancellation points will occur when a thread is executing any of the following functions:

*accept*( )
*connect*( )
*recv*( )
*recvfrom*( )
*recvmsg*( )
*send*( )
*sendmsg*( )
*sendto*( )

*t_close*( )
*t_connect*( )
*t_listen*( )
*t_rcv*( )
*t_rcvconnect*( )
*t_rcvrel*( )
*t_rcvreldata*( )
*t_rcvudata*( )
*t_rcvv*( )
*t_rcvvudata*( )
*t_snd*( )

*t_sndrel*( )
*t_sndreldata*( )
*t_sndudata*( )
*t_sndv*( )
*t_sndvudata*( )

A cancellation point may also occur when a thread is executing any of the following functions:

*endhostent*( )
*endnetent*( )
*endprotoent*( )
*endservent*( )
*gethostbyaddr*( )
*gethostbyname*( )
*gethostent*( )
*gethostname*( )
*getnetbyaddr*( )
*getnetbyname*( )
*getnetent*( )
*getprotobynumber*( )
*getprotobyname*( )
*getprotoent*( )
*getservbyport*( )
*getservbyname*( )
*getservent*( )
*sethostent*( )
*setnetent*( )
*setprotoent*( )
*setservent*( )

An implementation will not introduce cancellation points into any other function specified in this document.

See the referenced **XSH**, Section 2.8 for further information.

## 1.7 Relationship to Emerging Formal Standards

The IEEE 1003.1g standards committee is also developing interfaces to XTI and Sockets. X/Open is actively involved in the work of this committee.

*Technical Standard*

# Networking Services (XNS) Issue 5.2

# Part 2: Sockets

*The Open Group*

Chapter 2

# Sockets Interfaces

Support for the Sockets interfaces as defined in this Part 2 of the XNS Technical Standard is mandatory.

This chapter gives an overview of the Sockets interfaces and includes functions, macros and external variables to support portability at the C-language source level.

## 2.1 Sockets Overview

All network protocols are associated with a specific protocol family. A protocol family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services can include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family can support multiple methods of addressing, though the current protocol implementations do not. A protocol family normally comprises a number of protocols, one per socket type. It is not required that a protocol family support all socket types. A protocol family can contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in the manual page for the *socket*( ) function. A specific protocol can be accessed either by creating a socket of the appropriate type and protocol family, or by requesting the protocol explicitly when creating a socket. Protocols normally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family and network architecture. Certain semantics of the basic socket abstractions are protocol specific. All protocols are expected to support the basic model for their particular socket type, but can, in addition, provide nonstandard facilities or extensions to a mechanism. For example, a protocol supporting the SOCK_STREAM abstraction can allow more than one byte of out-of-band data to be transmitted per out-of-band message.

Addressing
: Associated with each address family is an address format. All network addresses adhere to a general structure, called a **sockaddr**. The length of the structure varies according to the address family.

Routing
: Sockets provides packet routing facilities. A routing information database is maintained, which is used in selecting the appropriate network interface when transmitting packets.

Interfaces
: Each network interface in a system corresponds to a path through which messages can be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface do not.

Chapter 6 on page 105, Chapter 7 on page 107, and Chapter 8 on page 111, respectively describe the use of sockets for local UNIX connections, for Internet protocols based on IPv4, and for Internet protocols based on IPv6.

**NAME**

accept — accept a new connection on a socket

**SYNOPSIS**

```
#include <sys/socket.h>

int accept (int socket, struct sockaddr *address,
    socklen_t *address_len);
```

**DESCRIPTION**

The *accept*( ) function extracts the first connection on the queue of pending connections, creates a new socket with the same socket type protocol and address family as the specified socket, and allocates a new file descriptor for that socket.

The function takes the following arguments:

*socket* Specifies a socket that was created with *socket*( ), has been bound to an address with *bind*( ), and has issued a successful call to *listen*( ).

*address* Either a null pointer, or a pointer to a **sockaddr** structure where the address of the connecting socket will be returned.

*address_len* Points to a **socklen_t** which on input specifies the length of the supplied **sockaddr** structure, and on output specifies the length of the stored address.

If *address* is not a null pointer, the address of the peer for the accepted connection is stored in the **sockaddr** structure pointed to by *address*, and the length of this address is stored in the object pointed to by *address_len*.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address will be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by *address* is unspecified.

If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file descriptor for the socket, *accept*( ) will block until a connection is present. If the *listen*( ) queue is empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket, *accept*( ) will fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

The accepted socket cannot itself accept more connections. The original socket remains open and can accept more connections.

**RETURN VALUE**

Upon successful completion, *accept*( ) returns the non-negative file descriptor of the accepted socket. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *accept*( ) function will fail if:

[EAGAIN] or [EWOULDBLOCK]
O_NONBLOCK is set for the socket file descriptor and no connections are present to be accepted.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNABORTED] A connection has been aborted.

[EFAULT] The *address* or *address_len* parameter can not be accessed or written.

| | |
|---|---|
| [EINTR] | The *accept*( ) function was interrupted by a signal that was caught before a valid connection arrived. |
| [EINVAL] | The *socket* is not accepting connections. |
| [EMFILE] | {OPEN_MAX} file descriptors are currently open in the calling process. |
| [ENFILE] | The maximum number of file descriptors in the system are already open. |
| [ENOTSOCK] | The *socket* argument does not refer to a socket. |
| [EOPNOTSUPP] | The socket type of the specified socket does not support accepting connections. |

The *accept*( ) function may fail if:

| | |
|---|---|
| [ENOBUFS] | No buffer space is available. |
| [ENOMEM] | There was insufficient memory available to complete the operation. |
| [ENOSR] | There was insufficient STREAMS resources available to complete the operation. |
| [EPROTO] | A protocol error has occurred; for example, the STREAMS protocol stack has not been initialised. |

**APPLICATION USAGE**

When a connection is available, *select*( ) will indicate that the file descriptor for the socket is ready for reading.

**SEE ALSO**

*bind*( ), *connect*( ), *listen*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

bind — bind a name to a socket

**SYNOPSIS**

```
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address,
     socklen_t address_len);
```

**DESCRIPTION**

The *bind*( ) function assigns an *address* to an unnamed socket. Sockets created with the *socket*( ) function are initially unnamed; they are identified only by their address family.

The function takes the following arguments:

*socket* Specifies the file descriptor of the socket to be bound.

*address* 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.

*address_len* Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

The socket in use may require the process to have appropriate privileges to use the *bind*( ) function.

**RETURN VALUE**

Upon successful completion, *bind*( ) returns 0. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *bind*( ) function will fail if:

[EADDRINUSE] The specified address is already in use.

[EADDRNOTAVAIL] The specified address is not available from the local machine.

[EAFNOSUPPORT] The specified address is not a valid address for the address family of the specified socket.

[EBADF] The *socket* argument is not a valid file descriptor.

[EFAULT} The *address* argument can not be accessed.

[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.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The socket type of the specified socket does not support binding to an address.

If the address family of the socket is AF_UNIX, then *bind*( ) will fail if:

[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.

[EDESTADDRREQ] or [EISDIR]
The *address* argument is a null pointer.

| | |
|---|---|
| [EIO] | An I/O error occurred. |
| [ELOOP] | Too many symbolic links were encountered in translating the pathname in *address*. |
| [ENAMETOOLONG] | A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters. |
| [ENOENT] | A component of the pathname does not name an existing file or the pathname is an empty string. |
| [ENOTDIR] | A component of the path prefix of the pathname in *address* is not a directory. |
| [EROFS] | The name would reside on a read-only filesystem. |

The *bind*( ) function may fail if:

| | |
|---|---|
| [EACCES] | The specified address is protected and the current user does not have permission to bind to it. |
| [EINVAL] | The *address_len* argument is not a valid length for the address family. |
| [EISCONN] | The socket is already connected. |
| [ENAMETOOLONG] | Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |
| [ENOBUFS] | Insufficient resources were available to complete the call. |
| [ENOSR] | There were insufficient STREAMS resources for the operation to complete. |

**APPLICATION USAGE**

An application program can retrieve the assigned socket name with the *getsockname*( ) function.

**SEE ALSO**

*connect*( ), *getsockname*( ), *listen*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

close — close a file descriptor

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**DESCRIPTION**

If *fildes* refers to a socket, *close*( ) causes the socket to be destroyed. If the socket is connection-mode, and the SO_LINGER option is set for the socket with non-zero linger time, and the socket has untransmitted data, then *close*( ) will block for up to the current linger interval until all data is transmitted.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

connect — connect a socket

**SYNOPSIS**

```
#include <sys/socket.h>

int connect(int socket, const struct sockaddr *address,
    socklen_t address_len);
```

**DESCRIPTION**

The *connect*( ) function requests a connection to be made on a socket. The function takes the following arguments:

| | |
|---|---|
| *socket* | Specifies the file descriptor associated with the socket. |
| *address* | Points to a **sockaddr** structure containing the peer address. The length and format of the address depend on the address family of the socket. |
| *address_len* | Specifies the length of the **sockaddr** structure pointed to by the *address* argument. |

If the socket has not already been bound to a local address, *connect*( ) will bind it to an address which, unless the socket's address family is AF_UNIX, is an unused local address.

If the initiating socket is not connection-mode, then *connect*( ) sets the socket's peer address, but no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all datagrams are sent on subsequent *send*( ) calls, and limits the remote sender for subsequent *recv*( ) calls. If *address* is a null address for the protocol, the socket's peer address will be reset.

If the initiating socket is connection-mode, then *connect*( ) attempts to establish a connection to the address specified by the *address* argument.

If the connection cannot be established immediately and O_NONBLOCK is not set for the file descriptor for the socket, *connect*( ) will block for up to an unspecified timeout interval until the connection is established. If the timeout interval expires before the connection is established, *connect*( ) will fail and the connection attempt will be aborted. If *connect*( ) is interrupted by a signal that is caught while blocked waiting to establish a connection, *connect*( ) will fail and set *errno* to [EINTR], but the connection request will not be aborted, and the connection will be established asynchronously.

If the connection cannot be established immediately and O_NONBLOCK is set for the file descriptor for the socket, *connect*( ) will fail and set *errno* to [EINPROGRESS], but the connection request will not be aborted, and the connection will be established asynchronously. Subsequent calls to *connect*( ) for the same socket, before the connection is established, will fail and set *errno* to [EALREADY].

When the connection has been established asynchronously, *select*( ) and *poll*( ) will indicate that the file descriptor for the socket is ready for writing.

The socket in use may require the process to have appropriate privileges to use the *connect*( ) function.

**RETURN VALUE**

Upon successful completion, *connect*( ) returns 0. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *connect*( ) function will fail if:

[EADDRNOTAVAIL] The specified address is not available from the local machine.

[EAFNOSUPPORT] The specified address is not a valid address for the address family of the specified socket.

[EALREADY] A connection request is already in progress for the specified socket.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNREFUSED] The target address was not listening for connections or refused the connection request.

[EFAULT] The address parameter can not be accessed.

[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.

[EINTR] The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection will be established asynchronously.

[EISCONN] The specified socket is connection-mode and is already connected.

[ENETUNREACH] No route to the network is present.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EPROTOTYPE] The specified address has a different type than the socket bound to the specified peer address.

[ETIMEDOUT] The attempt to connect timed out before a connection was made.

If the address family of the socket is AF_UNIX, then *connect*( ) will fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ELOOP] Too many symbolic links were encountered in translating the pathname in *address*.

[ENAMETOOLONG] A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.

[ENOENT] A component of the pathname does not name an existing file or the pathname is an empty string.

[ENOTDIR] A component of the path prefix of the pathname in *address* is not a directory.

The *connect*( ) function may fail if:

[EACCES] Search permission is denied for a component of the path prefix; or write access to the named socket is denied.

[EADDRINUSE] Attempt to establish a connection that uses addresses that are already in use.

[ECONNRESET] Remote host reset the connection request.

[EHOSTUNREACH] The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

| | |
|---|---|
| [EINVAL] | The *address_len* argument is not a valid length for the address family; or invalid address family in sockaddr structure. |
| [ENAMETOOLONG] | Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |
| [ENETDOWN] | The local interface used to reach the destination is down. |
| [ENOBUFS] | No buffer space is available. |
| [ENOSR] | There were insufficient STREAMS resources available to complete the operation. |
| [EOPNOTSUPP] | The socket is listening and can not be connected. |

**APPLICATION USAGE**

If *connect*( ) fails, the state of the socket is unspecified. Portable applications should close the file descriptor and create a new socket before attempting to reconnect.

**SEE ALSO**

*accept*( ), *bind*( ), *close*( ), *getsockname*( ), *poll*( ), *select*( ), *send*( ), *shutdown*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

fcntl — file control

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**DESCRIPTION**

The following additional values for *cmd* are defined in **<fcntl.h>**:

F_GETOWN — If *fildes* 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 *fildes* does not refer to a socket, the results are unspecified.

F_SETOWN — If *fildes* 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, *arg*, taken as type **int**. Positive values indicate a process ID; negative values, other than –1, indicate a process group ID. If *fildes* does not refer to a socket, the results are unspecified.

**RETURN VALUE**

Upon successful completion, the value returned depends on *cmd* as follows:

F_GETOWN — Value of the socket owner process or process group; this will not be –1.

F_SETOWN — Value other than –1.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

fgetpos — get current file position information

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**ERRORS**

The *fgetpos*( ) function may fail if:

[ESPIPE] The file descriptor underlying *stream* is associated with a socket.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

fsetpos — set current file position

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**ERRORS**

The *fsetpos*( ) function may fail if:

[ESPIPE] The file descriptor underlying *stream* is associated with a socket.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

ftell — return a file offset in a stream

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**ERRORS**

The *ftell*( ) function may fail if:

[ESPIPE] The file descriptor underlying *stream* is associated with a socket.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

getpeername — get the name of the peer socket

**SYNOPSIS**

```
#include <sys/socket.h>

int getpeername(int socket, struct sockaddr *address,
    socklen_t *address_len);
```

**DESCRIPTION**

The *getpeername*( ) function retrieves the peer address of the specified socket, stores this address in the **sockaddr** structure pointed to by the *address* argument, and stores the length of this address in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address will be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by *address* is unspecified.

**RETURN VALUE**

Upon successful completion, 0 is returned. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *getpeername*( ) function will fail if:

| | |
|---|---|
| [EBADF] | The *socket* argument is not a valid file descriptor. |
| [EFAULT] | The *address* or *address_len* parameter can not be accessed or written. |
| [EINVAL] | The socket has been shut down. |
| [ENOTCONN] | The socket is not connected or otherwise has not had the peer prespecified. |
| [ENOTSOCK] | The *socket* argument does not refer to a socket. |
| [EOPNOTSUPP] | The operation is not supported for the socket protocol. |

The *getpeername*( ) function may fail if:

| | |
|---|---|
| [ENOBUFS] | Insufficient resources were available in the system to complete the call. |
| [ENOSR] | There were insufficient STREAMS resources available for the operation to complete. |

**SEE ALSO**

*accept*( ), *bind*( ), *getsockname*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

getsockname — get the socket name

**SYNOPSIS**

```
#include <sys/socket.h>

int getsockname(int socket, struct sockaddr *address,
    socklen_t *address_len);
```

**DESCRIPTION**

The *getsockname*( ) function retrieves the locally-bound name of the specified socket, stores this address in the **sockaddr** structure pointed to by the *address* argument, and stores the length of this address in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address will be truncated.

If the socket has not been bound to a local name, the value stored in the object pointed to by *address* is unspecified.

**RETURN VALUE**

Upon successful completion, 0 is returned, the *address* argument points to the address of the socket, and the *address_len* argument points to the length of the address. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *getsockname*( ) function will fail:

| | |
|---|---|
| [EBADF] | The *socket* argument is not a valid file descriptor. |
| [EFAULT] | The *address* or *address_len* parameter can not be accessed or written. |
| [ENOTSOCK] | The *socket* argument does not refer to a socket. |
| [EOPNOTSUPP] | The operation is not supported for this socket's protocol. |

The *getsockname*( ) function may fail if:

| | |
|---|---|
| [EINVAL] | The socket has been shut down. |
| [ENOBUFS] | Insufficient resources were available in the system to complete the call. |
| [ENOSR] | There were insufficient STREAMS resources available for the operation to complete. |

**SEE ALSO**

*accept*( ), *bind*( ), *getpeername*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

getsockopt — get the socket options

**SYNOPSIS**

```
#include <sys/socket.h>

int getsockopt(int socket, int level, int option_name,
     void *option_value, socklen_t *option_len);
```

**DESCRIPTION**

The *getsockopt*( ) function retrieves the value for the option specified by the *option_name* argument for the socket specified by the *socket* argument. If the size of the option value is greater than *option_len*, the value stored in the object pointed to by the *option_value* argument will be silently truncated. Otherwise, the object pointed to by the *option_len* argument will be modified to indicate the actual length of the value.

The *level* argument specifies the protocol level at which the option resides. To retrieve options at the socket level, specify the *level* argument as SOL_SOCKET. To retrieve options at other levels, supply the appropriate level identifier for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP (Transmission Control Protocol), set *level* to IPPROTO_TCP as defined in the **<netinet/in.h>** header.

The socket in use may require the process to have appropriate privileges to use the *getsockopt*( ) function.

The *option_name* argument specifies a single option to be retrieved. It can be one of the following values defined in **<sys/socket.h>**:

SO_DEBUG — Reports whether debugging information is being recorded. This option stores an **int** value. This is a boolean option.

SO_ACCEPTCONN — Reports whether socket listening is enabled. This option stores an **int** value. This is a boolean option.

SO_BROADCAST — Reports whether transmission of broadcast messages is supported, if this is supported by the protocol. This option stores an **int** value. This is a boolean option.

SO_REUSEADDR — Reports whether the rules used in validating addresses supplied to *bind*( ) should allow reuse of local addresses, if this is supported by the protocol. This option stores an **int** value. This is a boolean option.

SO_KEEPALIVE — Reports whether connections are kept active with periodic transmission of messages, if this is supported by the protocol.

If the connected socket fails to respond to these messages, the connection is broken and processes writing to that socket are notified with a SIGPIPE signal. This option stores an **int** value.

This is a boolean option.

SO_LINGER — Reports whether the socket lingers on *close*( ) if data is present. If SO_LINGER is set, the system blocks the process during *close*( ) until it can transmit the data or until the end of the interval indicated by the **l_linger** member, whichever comes first. If SO_LINGER is not specified, and *close*( ) is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option stores a **linger** structure.

SO_OOBINLINE Reports whether the socket leaves received out-of-band data (data marked urgent) in line. This option stores an **int** value. This is a boolean option.

SO_SNDBUF Reports send buffer size information. This option stores an **int** value.

SO_RCVBUF Reports receive buffer size information. This option stores an **int** value.

SO_ERROR Reports information about error status and clears it. This option stores an **int** value.

SO_TYPE Reports the socket type. This option stores an **int** value.

SO_DONTROUTE Reports whether outgoing messages bypass the standard routing facilities. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option stores an **int** value. This is a boolean option.

SO_RCVLOWAT Reports the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned, e.g. out of band data). This option stores an **int** value. Note that not all implementations allow this option to be retrieved.

SO_RCVTIMEO Reports the timeout value for input operations. This option stores a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it returns with a partial count or errno set to [EAGAIN] or [EWOULDBLOCK] if no data were received. The default for this option is zero, which indicates that a receive operation will not time out. Note that not all implementations allow this option to be retrieved.

SO_SNDLOWAT Reports the minimum number of bytes to process for socket output operations. Non-blocking output operations will process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option stores an **int** value. Note that not all implementations allow this option to be retrieved.

SO_SNDTIMEO Reports the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it returns with a partial count or with errno set to [EAGAIN] or [EWOULDBLOCK] if no data were sent. The default for this option is zero, which indicates that a send operation will not time out. The option stores a **timeval** structure. Note that not all implementations allow this option to be retrieved.

For boolean options, a zero value indicates that the option is disabled and a non-zero value indicates that the option is enabled.

Options at other protocol levels vary in format and name.

The socket in use may require the process to have appropriate privileges to use the *getsockopt*( ) function.

**RETURN VALUE**

Upon successful completion, *getsockopt*( ) returns 0. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *getsockopt*( ) function will fail if:

| | |
|---|---|
| [EBADF] | The *socket* argument is not a valid file descriptor. |
| [EFAULT] | The *option_value* or *option_len* parameter can not be accessed or written. |
| [EINVAL] | The specified option is invalid at the specified socket level. |
| [ENOPROTOOPT] | The option is not supported by the protocol. |
| [ENOTSOCK] | The *socket* argument does not refer to a socket. |

The *getsockopt*( ) function may fail if:

| | |
|---|---|
| [EACCES] | The calling process does not have the appropriate privileges. |
| [EINVAL] | The socket has been shut down. |
| [ENOBUFS] | Insufficient resources are available in the system to complete the call. |
| [ENOSR] | There were insufficient STREAMS resources available for the operation to complete. |

**SEE ALSO**

*bind*( ), *close*( ), *endprotoent*( ), *setsockopt*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

if_freenameindex — free memory allocated by *ifnameindex*( )

**SYNOPSIS**

```
#include <net/if.h>

void if_freenameindex(struct if_nameindex *ptr);
```

**DESCRIPTION**

Argument *if_nameindex* must be a pointer that was returned by *if_nameindex*( ). Function *if_freenameindex*( ) frees the memory allocated by *if_nameindex*( ). After *if_freenameindex*( ) has been called, the application should not use the array of which *ptr* is the address.

**ERRORS**

[EFAULT] Argument *if_nameindex* is not a pointer that was returned by *if_nameindex*( ). Note that implementations need not always detect this fault. If an implementation does not do so, the results are unspecified.

**SEE ALSO**

*getsockopt*( ), i*f_indextoname*( ), *if_nameindex*( ), *if_nametoindex*( ), *setsockopt*( ), **<net/if.h>**.

**CHANGE HISTORY**

First released in Issue 5.2.

**NAME**

if_indextoname — map an interface index to its corresponding name

**SYNOPSIS**

```
#include <net/if.h>

char *if_indextoname(unsigned int ifindex, char *ifname);
```

**DESCRIPTION**

When this function is called, *ifname* must point to a buffer of at least IFNAMSIZ bytes. The function places in this buffer the name of the interface with index *ifindex*.

**RETURN VALUE**

If *ifindex* is an interface index then the function returns the value supplied in *ifname*, which points to a buffer now containing the interface name. Otherwise the function returns a NULL pointer.

**ERRORS**

[EFAULT] The buffer pointed to by *ifname* can not be accessed or written.

**SEE ALSO**

*getsockopt*( ), *if_freenameindex*( ), *if_nameindex*( ), *if_nametoindex*( ), *setsockopt*( ), **<net/if.h>**.

**CHANGE HISTORY**

First released in Issue 5.2.

**NAME**

if_nameindex — return all interface names and indexes

**SYNOPSIS**

```
#include <net/if.h>

struct if_nameindex *if_nameindex(void);
```

**DESCRIPTION**

This function returns an array of *if_nameindex* structures, one structure per interface. The end of the array is indicated by a structure with an *if_index* field of zero and an *if_name* field of NULL.

Applications should call *if_freenameindex*( ) to release the memory that may be dynamically allocated by this function, after they have finished using it.

**RETURN VALUE**

Array of structures identifying local interfaces. A NULL pointer is returned upon an error, with errno set to indicate the nature of the error.

**ERRORS**

[ENOBUFS] Insufficient resources are available in the system to complete the call.

**SEE ALSO**

*getsockopt*( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nametoindex*( ), *setsockopt*( ), **<net/if.h>**

**CHANGE HISTORY**

First released in Issue 5.2.

**NAME**

if_nametoindex — map an interface name to its corresponding index

**SYNOPSIS**

```
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);
```

**DESCRIPTION**

Returns the interface index corresponding to name *ifname*.

**RETURN VALUE**

The corresponding index if *ifname* is the name of an interface; zero otherwise.

**ERRORS**

[EFAULT] The buffer pointed to by *ifname* can not be accessed.

**SEE ALSO**

*getsockopt*( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nameindex*( ), *setsockopt*( ), **<net/if.h>**.

**CHANGE HISTORY**

First released in Issue 5.2.

**NAME**

listen — listen for socket connections and limit the queue of incoming connections

**SYNOPSIS**

```
#include <sys/socket.h>

int listen(int socket, int backlog);
```

**DESCRIPTION**

The *listen*( ) function marks a connection-mode socket, specified by the socket argument, as accepting connections.

The *backlog* argument provides a hint to the implementation which the implementation will use to limit the number of outstanding connections in the socket's listen queue. Normally, a larger backlog argument value will result in a larger or equal length of the listen queue.

The implementation may include incomplete connections in its listen queue. The limits on the number of incomplete connections and completed connections queued may be different.

The implementation may have an upper limit on the length of the listen queue - either global or per accepting socket. If *backlog* exceeds this limit, the length of the listen queue is set to the limit.

If *listen*( ) is called with a *backlog* argument value that is less than 0, the function behaves as if it had been called with a *backlog* argument value of 0.

A *backlog* argument of 0 may allow the socket to accept connections, in which case the length of the listen queue may be set to an implementation-dependent minimum value.

The socket in use may require the process to have appropriate privileges to use the *listen*( ) function.

**RETURN VALUE**

Upon successful completions, *listen*( ) returns 0. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *listen*( ) function will fail if:

| | |
|---|---|
| [EBADF] | The *socket* argument is not a valid file descriptor. |
| [EDESTADDRREQ] | The socket is not bound to a local address, and the protocol does not support listening on an unbound socket. |
| [EINVAL] | The *socket* is already connected. |
| [ENOTSOCK] | The *socket* argument does not refer to a socket. |
| [EOPNOTSUPP] | The socket protocol does not support *listen*( ). |

The *listen*( ) function may fail if:

| | |
|---|---|
| [EACCES] | The calling process does not have the appropriate privileges. |
| [EINVAL] | The *socket* has been shut down. |
| [ENOBUFS] | Insufficient resources are available in the system to complete the call. |

**SEE ALSO**

*accept*( ), *connect*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

lseek — move read/write file offset

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**ERRORS**

The *lseek*( ) function will fail if:

[ESPIPE] The file descriptor underlying *stream* is associated with a socket.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

poll — input/output multiplexing

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**DESCRIPTION**

The *poll*( ) function supports sockets.

A file descriptor for a socket that is listening for connections will indicate that it is ready for reading, once connections are available. A file descriptor for a socket that is connecting asynchronously will indicate that it is ready for writing, once a connection has been established.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

read, readv — read from file

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**DESCRIPTION**

If *fildes* refers to a socket, *read*( ) is equivalent to *recv*( ) with no flags set.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

recv — receive a message from a connected socket

**SYNOPSIS**

```
#include <sys/socket.h>

ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

**DESCRIPTION**

The *recv*( ) function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data. The function takes the following arguments:

*socket* Specifies the socket file descriptor.

*buffer* Points to a buffer where the message should be stored.

*length* Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

*flags* Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

| | |
|---|---|
| MSG_PEEK | Peeks at an incoming message. The data is treated as unread and the next *recv*( ) or similar function will still return this data. |
| MSG_OOB | Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific. |
| 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, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket. |

The *recv*( ) function returns the length of the message written to the buffer pointed to by the *buffer* argument. For message-based sockets such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message must be read in a single operation. If a message is too long to fit in the supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes are discarded. For stream-based sockets such as SOCK_STREAM, message boundaries are ignored. In this case, data is returned to the user as soon as it becomes available, and no data is discarded.

If the MSG_WAITALL flag is not set, data will be returned only up to the end of the first message.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recv*( ) blocks until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recv*( ) fails and sets *errno* to [EAGAIN] or [EWOULDBLOCK].

**RETURN VALUE**

Upon successful completion, *recv*( ) returns the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recv*( ) returns 0. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *recv*( ) function will fail if:

[EAGAIN] or [EWOULDBLOCK]
: 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.

[EBADF]
: The *socket* argument is not a valid file descriptor.

[ECONNRESET]
: A connection was forcibly closed by a peer.

[EFAULT]
: The *buffer* parameter can not be accessed or written.

[EINTR]
: The *recv*( ) function was interrupted by a signal that was caught, before any data was available.

[EINVAL]
: The MSG_OOB flag is set and no out-of-band data is available.

[ENOTCONN]
: A receive is attempted on a connection-mode socket that is not connected.

[ENOTSOCK]
: The *socket* argument does not refer to a socket.

[EOPNOTSUPP]
: The specified flags are not supported for this socket type or protocol.

[ETIMEDOUT]
: The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recv*( ) function may fail if:

[EIO]
: An I/O error occurred while reading from or writing to the file system.

[ENOBUFS]
: Insufficient resources were available in the system to perform the operation.

[ENOMEM]
: Insufficient memory was available to fulfill the request.

[ENOSR]
: There were insufficient STREAMS resources available for the operation to complete.

**APPLICATION USAGE**

The *recv*( ) function is identical to *recvfrom*( ) with a zero *address_len* argument, and to *read*( ) if no flags are used.

The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

**SEE ALSO**

*poll*( ), *read*( ), *recvmsg*( ), *recvfrom*( ), *select*( ), *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), *write*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

recvfrom — receive a message from a socket

**SYNOPSIS**

```
#include <sys/socket.h>

ssize_t recvfrom(int socket, void *buffer, size_t length, int flags,
    struct sockaddr *address, socklen_t *address_len);
```

**DESCRIPTION**

The *recvfrom*( ) function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

The function takes the following arguments:

*socket* Specifies the socket file descriptor.

*buffer* Points to the buffer where the message should be stored.

*length* Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

*flags* Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

- MSG_PEEK — Peeks at an incoming message. The data is treated as unread and the next *recvfrom*( ) or similar function will still return this data.
- MSG_OOB — Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
- 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, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

*address* A null pointer, or points to a **sockaddr** structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket.

*address_len* Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

The *recvfrom*( ) function returns the length of the message written to the buffer pointed to by the *buffer* argument. For message-based sockets such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message must be read in a single operation. If a message is too long to fit in the supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes are discarded. For stream-based sockets such as SOCK_STREAM, message boundaries are ignored. In this case, data is returned to the user as soon as it becomes available, and no data is discarded.

If the MSG_WAITALL flag is not set, data will be returned only up to the end of the first message.

Not all protocols provide the source address for messages. If the *address* argument is not a null pointer and the protocol provides the source address of messages, the source address of the

received message is stored in the **sockaddr** structure pointed to by the *address* argument, and the length of this address is stored in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address will be truncated.

If the *address* argument is not a null pointer and the protocol does not provide the source address of messages, the the value stored in the object pointed to by *address* is unspecified.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recvfrom*( ) blocks until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recvfrom*( ) fails and sets *errno* to [EAGAIN] or [EWOULDBLOCK].

**RETURN VALUE**

Upon successful completion, *recvfrom*( ) returns the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvfrom*( ) returns 0. Otherwise the function returns –1 and sets *errno* to indicate the error.

**ERRORS**

The *recvfrom*( ) function will fail if:

[EAGAIN] or [EWOULDBLOCK]
: 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.

[EBADF]
: The *socket* argument is not a valid file descriptor.

[ECONNRESET]
: A connection was forcibly closed by a peer.

[EFAULT]
: The *buffer*, *address* or *address_len* parameter can not be accessed or written.

[EINTR]
: A signal interrupted *recvfrom*( ) before any data was available.

[EINVAL]
: The MSG_OOB flag is set and no out-of-band data is available.

[ENOTCONN]
: A receive is attempted on a connection-mode socket that is not connected.

[ENOTSOCK]
: The *socket* argument does not refer to a socket.

[EOPNOTSUPP]
: The specified flags are not supported for this socket type.

[ETIMEDOUT]
: The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recvfrom*( ) function may fail if:

[EIO]
: An I/O error occurred while reading from or writing to the file system.

[ENOBUFS]
: Insufficient resources were available in the system to perform the operation.

[ENOMEM]
: Insufficient memory was available to fulfill the request.

[ENOSR]
: There were insufficient STREAMS resources available for the operation to complete.

**APPLICATION USAGE**

The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

**SEE ALSO**

*poll*( ), *read*( ), *recv*( ), *recvmsg*( ), *select*( ) *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), *write*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

recvmsg — receive a message from a socket

**SYNOPSIS**

```
#include <sys/socket.h>

ssize_t recvmsg(int socket, struct msghdr *message, int flags);
```

**DESCRIPTION**

The *recvmsg*( ) function receives a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

The function takes the following arguments:

*socket* Specifies the socket file descriptor.

*message* 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.

*flags* Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

| | |
|---|---|
| MSG_OOB | Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific. |
| MSG_PEEK | Peeks at the incoming message. |
| 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, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket. |

The *recvmsg*( ) function receives messages from unconnected or connected sockets and returns the length of the message.

The *recvmsg*( ) function returns the total length of the message. For message-based sockets such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message must be read in a single operation. If a message is too long to fit in the supplied buffers, and MSG_PEEK is not set in the *flags* argument, the excess bytes are discarded, and MSG_TRUNC is set in the **msg_flags** member of the **msghdr** structure. For stream-based sockets such as SOCK_STREAM, message boundaries are ignored. In this case, data is returned to the user as soon as it becomes available, and no data is discarded.

If the MSG_WAITALL flag is not set, data will be returned only up to the end of the first message.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recvfrom*( ) blocks until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recvfrom*( ) function fails and sets *errno* to [EAGAIN] or [EWOULDBLOCK].

In the **msghdr** structure, the **msg_name** and **msg_namelen** members specify the source address if the socket is unconnected. If the socket is connected, the **msg_name** and **msg_namelen**

members are ignored. The **msg_name** member may be a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* fields are used to specify where the received data will be stored. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* must be set to the dimension of this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Each storage area indicated by *msg_iov* is filled with received data in turn until all of the received data is stored or all of the areas have been filled.

On successful completion, the **msg_flags** member of the message header is the bitwise-inclusive OR of all of the following flags that indicate conditions detected for the received message:.

| | |
|---|---|
| MSG_EOR | End of record was received (if supported by the protocol). |
| MSG_OOB | Out-of-band data was received. |
| MSG_TRUNC | Normal data was truncated. |
| MSG_CTRUNC | Control data was truncated. |

**RETURN VALUE**

Upon successful completion, *recvmsg*( ) returns the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvmsg*( ) returns 0. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *recvmsg*( ) function will fail if:

| | |
|---|---|
| [EAGAIN] or [EWOULDBLOCK] | 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. |
| [EBADF] | The *socket* argument is not a valid open file descriptor. |
| [ECONNRESET] | A connection was forcibly closed by a peer. |
| [EFAULT] | The *message* parameter, or storage pointed to by the *msg_name*, *msg_control* or *msg_iov* fields of the *message* parameter, or storage pointed to by the **iovec** structures pointed to by the *msg_iov* field can not be accessed or written. |
| [EINTR] | This function was interrupted by a signal before any data was available. |
| [EINVAL] | The sum of the **iov_len** values overflows an **ssize_t**. or the MSG_OOB flag is set and no out-of-band data is available. |
| [EMSGSIZE] | The **msg_iovlen** member of the **msghdr** structure pointed to by *message* is less than or equal to 0, or is greater than {IOV_MAX}. |
| [ENOTCONN] | A receive is attempted on a connection-mode socket that is not connected. |
| [ENOTSOCK] | The *socket* argument does not refer to a socket. |
| [EOPNOTSUPP] | The specified flags are not supported for this socket type. |
| [ETIMEDOUT] | The connection timed out during connection establishment, or due to a transmission timeout on active connection. |

The *recvmsg*( ) function may fail if:

| | |
|---|---|
| [EIO] | An IO error occurred while reading from or writing to the file system. |
| [ENOBUFS] | Insufficient resources were available in the system to perform the operation. |
| [ENOMEM] | Insufficient memory was available to fulfill the request. |
| [ENOSR] | There were insufficient STREAMS resources available for the operation to complete. |

**APPLICATION USAGE**

The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

**SEE ALSO**

*poll*( ), *recv*( ), *recvfrom*( ), *select*( ), *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

select — synchronous I/O multiplexing

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**DESCRIPTION**

A file descriptor for a socket that is listening for connections will indicate that it is ready for reading, when connections are available. A file descriptor for a socket that is connecting asynchronously will indicate that it is ready for writing, when a connection has been established.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

send — send a message on a socket

**SYNOPSIS**

```
#include <sys/socket.h>

ssize_t send(int socket, const void *buffer, size_t length, int flags);
```

**DESCRIPTION**

*socket* Specifies the socket file descriptor.

*buffer* Points to the buffer containing the message to send.

*length* Specifies the length of the message in bytes.

*flags* Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:

| | |
|---|---|
| MSG_EOR | Terminates a record (if supported by the protocol) |
| 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. |

The *send*( ) function initiates transmission of a message from the specified socket to its peer. The *send*( ) function sends a message only when the socket is connected (including when the peer of a connectionless socket has been set via *connect*( )).

The length of the message to be sent is specified by the *length* argument. If the message is too long to pass through the underlying protocol, *send*( ) fails and no data is transmitted.

Successful completion of a call to *send*( ) does not guarantee delivery of the message. A return value of –1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does not have O_NONBLOCK set, *send*( ) blocks until space is available. If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does have O_NONBLOCK set, *send*( ) will fail. The *select*( ) and *poll*( ) functions can be used to determine when it is possible to send more data.

The socket in use may require the process to have appropriate privileges to use the *send*( ) function.

**RETURN VALUE**

Upon successful completion, *send*( ) returns the number of bytes sent. Otherwise, –1 is returned and *errno* is set to indicate the error.

**APPLICATION USAGE**

The *send*( ) function is identical to *sendto*( ) with a null pointer *dest_len* argument, and to *write*( ) if no flags are used.

**ERRORS**

The *send*( ) function will fail if:

[EAGAIN] or [EWOULDBLOCK]
The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EDESTADDRREQ] The socket is not connection-mode and no peer address is set.

[EFAULT] The *buffer* parameter can not be accessed.

[EINTR] A signal interrupted *send*( ) before any data was transmitted.

[EMSGSIZE] The message is too large be sent all at once, as the socket requires.

[ENOTCONN] The socket is not connected or otherwise has not had the peer prespecified.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

[EPIPE] The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.

The *send*( ) function may fail if:

[EACCES] The calling process does not have the appropriate privileges.

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENETDOWN] The local interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOSR] There were insufficient STREAMS resources available for the operation to complete.

**SEE ALSO**

*connect*( ), *getsockopt*( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *sendmsg*( ), *sendto*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

sendmsg — send a message on a socket using a message structure

**SYNOPSIS**

```
#include <sys/socket.h>

ssize_t sendmsg(int socket, const struct msghdr *message, int flags);
```

**DESCRIPTION**

The *sendmsg*( ) function sends a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message will be sent to the address specified by *msghdr*. If the socket is connection-mode, the destination address in *msghdr* is ignored.

The function takes the following arguments:

*socket* Specifies the socket file descriptor.

*message* 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.

*flags* Specifies the type of message transmission. The application may specify 0 or the following flag:

| | |
|---|---|
| MSG_EOR | Terminates a record (if supported by the protocol) |
| MSG_OOB | Sends out-of-band data on sockets that support out-of-bound data. The significance and semantics of out-of-band data are protocol-specific. |

The *msg_iov* and *msg_iovlen* fields of message specify zero or more buffers containing the data to be sent. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* must be set to the dimension of this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Some of these sizes can be zero. The data from each storage area indicated by *msg_iov* is sent in turn.

Successful completion of a call to *sendmsg*( ) does not guarantee delivery of the message. A return value of –1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does not have O_NONBLOCK set, *sendmsg*( ) function blocks until space is available. If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does have O_NONBLOCK set, *sendmsg*( ) function will fail.

If the socket protocol supports broadcast and the specified address is a broadcast address for the socket protocol, *sendmsg*( ) will fail if the SO_BROADCAST option is not set for the socket.

The socket in use may require the process to have appropriate privileges to use the *sendmsg*( ) function.

**RETURN VALUE**

Upon successful completion, *sendmsg*( ) function returns the number of bytes sent. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *sendmsg*( ) function will fail if:

[EAGAIN] or [EWOULDBLOCK]
The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.

[EAFNOSUPPORT] Addresses in the specified address family cannot be used with this socket.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EFAULT] The *message* parameter, or storage pointed to by the *msg_name*, *msg_control* or *msg_iov* fields of the *message* parameter, or storage pointed to by the **iovec** structures pointed to by the *msg_iov* field can not be accessed.

[EINTR] A signal interrupted *sendmsg*( ) before any data was transmitted.

[EINVAL] The sum of the **iov_len** values overflows an **ssize_t**.

[EMSGSIZE] The message is too large to be sent all at once (as the socket requires), or the **msg_iovlen** member of the **msghdr** structure pointed to by *message* is less than or equal to 0 or is greater than {IOV_MAX}.

[ENOTCONN] The socket is connection-mode but is not connected.

[ENOTSOCK] The *socket* argument does not refer a socket.

[EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

[EPIPE] The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.

If the address family of the socket is AF_UNIX, then *sendmsg*( ) will fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ELOOP] Too many symbolic links were encountered in translating the pathname in the socket address.

[ENAMETOOLONG] A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.

[ENOENT] A component of the pathname does not name an existing file or the pathname is an empty string.

[ENOTDIR] A component of the path prefix of the pathname in the socket address is not a directory.

The *sendmsg*( ) function may fail if:

[EACCES] Search permission is denied for a component of the path prefix; or write access to the named socket is denied.

[EDESTADDRREQ] The socket is not connection-mode and does not have its peer address set, and no destination address was specified.

[EHOSTUNREACH] The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

[EIO] An I/O error occurred while reading from or writing to the file system.

[EISCONN] A destination address was specified and the socket is already connected.

[ENETDOWN] The local interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

[ENOSR] There were insufficient STREAMS resources available for the operation to complete.

If the address family of the socket is AF_UNIX, then *sendmsg*( ) may fail if:

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

**APPLICATION USAGE**

The *select*( ) and *poll*( ) functions can be used to determine when it is possible to send more data.

**SEE ALSO**

*getsockopt*( ), *poll*( ) *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendto*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

sendto — send a message on a socket

**SYNOPSIS**

```
#include <sys/socket.h>

ssize_t sendto(int socket, const void *message, size_t length, int flags,
      const struct sockaddr *dest_addr, socklen_t dest_len);
```

**DESCRIPTION**

The *sendto*( ) function sends a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message will be sent to the address specified by *dest_addr*. If the socket is connection-mode, *dest_addr* is ignored.

The function takes the following arguments:

*socket* Specifies the socket file descriptor.

*message* Points to a buffer containing the message to be sent.

*length* Specifies the size of the message in bytes.

*flags* Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:

- MSG_EOR Terminates a record (if supported by the protocol)
- 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.

*dest_addr* Points to a **sockaddr** structure containing the destination address. The length and format of the address depend on the address family of the socket.

*dest_len* Specifies the length of the **sockaddr** structure pointed to by the *dest_addr* argument.

If the socket protocol supports broadcast and the specified address is a broadcast address for the socket protocol, *sendto*( ) will fail if the SO_BROADCAST option is not set for the socket.

The *dest_addr* argument specifies the address of the target. The *length* argument specifies the length of the message.

Successful completion of a call to *sendto*( ) does not guarantee delivery of the message. A return value of –1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does not have O_NONBLOCK set, *sendto*( ) blocks until space is available. If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does have O_NONBLOCK set, *sendto*( ) will fail.

The socket in use may require the process to have appropriate privileges to use the *sendto*( ) function.

**RETURN VALUE**

Upon successful completion, *sendto*( ) returns the number of bytes sent. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *sendto*( ) function will fail if:

[EAFNOSUPPORT] Addresses in the specified address family cannot be used with this socket.

[EAGAIN] or [EWOULDBLOCK]
The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EFAULT] The *message* or *destaddr* parameter can not be accessed.

[EINTR] A signal interrupted *sendto*( ) before any data was transmitted.

[EMSGSIZE] The message is too large to be sent all at once, as the socket requires.

[ENOTCONN] The socket is connection-mode but is not connected.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

[EPIPE] The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.

If the address family of the socket is AF_UNIX, then *sendto*( ) will fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ELOOP] Too many symbolic links were encountered in translating the pathname in the socket address.

[ENAMETOOLONG] A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.

[ENOENT] A component of the pathname does not name an existing file or the pathname is an empty string.

[ENOTDIR] A component of the path prefix of the pathname in the socket address is not a directory.

The *sendto*( ) function may fail if:

[EACCES] Search permission is denied for a component of the path prefix; or write access to the named socket is denied.

[EDESTADDRREQ] The socket is not connection-mode and does not have its peer address set, and no destination address was specified.

[EHOSTUNREACH] The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

[EINVAL] The *dest_len* argument is not a valid length for the address family.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EISCONN] A destination address was specified and the socket is already connected. This error may or may not be returned for connection mode sockets.

[ENETDOWN] The local interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

[ENOSR] There were insufficient STREAMS resources available for the operation to complete.

If the address family of the socket is AF_UNIX, then *sendto*( ) may fail if:

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

**APPLICATION USAGE**

The *select*( ) and *poll*( ) functions can be used to determine when it is possible to send more data.

**SEE ALSO**

*getsockopt*( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendmsg*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

setsockopt — set the socket options

**SYNOPSIS**

```
#include <sys/socket.h>

int setsockopt(int socket, int level, int option_name,
    const void *option_value, socklen_t option_len);
```

**DESCRIPTION**

The *setsockopt*( ) function sets the option specified by the *option_name* argument, at the protocol level specified by the *level* argument, to the value pointed to by the *option_value* argument for the socket associated with the file descriptor specified by the *socket* argument.

The *level* argument specifies the protocol level at which the option resides. To set options at the socket level, specify the *level* argument as SOL_SOCKET. To set options at other levels, supply the appropriate lecel identifier for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP (Transport Control Protocol), set *level* to IPPROTO_TCP as defined in the **<netinet/in.h>** header.

The *option_name* argument specifies a single option to set. The *option_name* argument and any specified options are passed uninterpreted to the appropriate protocol module for interpretations. The <**sys/socket.h**> header defines the socket level options. The options are as follows:

SO_DEBUG — Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules. This option takes an **int** value. This is a boolean option.

SO_BROADCAST — Permits sending of broadcast messages, if this is supported by the protocol. This option takes an **int** value. This is a boolean option.

SO_REUSEADDR — Specifies that the rules used in validating addresses supplied to *bind*( ) should allow reuse of local addresses, if this is supported by the protocol. This option takes an **int** value. This is a boolean option.

SO_KEEPALIVE — Keeps connections active by enabling the periodic transmission of messages, if this is supported by the protocol. This option takes an **int** value.

If the connected socket fails to respond to these messages, the connection is broken and processes writing to that socket are notified with a SIGPIPE signal.

This is a boolean option.

SO_LINGER — Lingers on a *close*( ) if data is present. This option controls the action taken when unsent messages queue on a socket and *close*( ) is performed. If SO_LINGER is set, the system blocks the process during *close*( ) until it can transmit the data or until the time expires. If SO_LINGER is not specified, and *close*( ) is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option takes a **linger** structure, as defined in the <**sys/socket.h**> header, to specify the state of the option and linger interval.

SO_OOBINLINE — Leaves received out-of-band data (data marked urgent) in line. This option takes an **int** value. This is a boolean option.

SO_SNDBUF Sets send buffer size. This option takes an **int** value.

SO_RCVBUF Sets receive buffer size. This option takes an **int** value.

SO_DONTROUTE Requests that outgoing messages bypass the standard routing facilities. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option takes an **int** value. This is a boolean option.

SO_RCVLOWAT Sets the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned, e.g. out of band data). This option takes an **int** value. Note that not all implementations allow this option to be set.

SO_RCVTIMEO Sets the timeout value that specifies the maximum amount of time an input function waits until it completes. It accepts a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it returns with a partial count or *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data were received. The default for this option is zero, which indicates that a receive operation will not time out. This option takes a **timeval** structure. Note that not all implementations allow this option to be set.

SO_SNDLOWAT Sets the minimum number of bytes to process for socket output operations. Non-blocking output operations will process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option takes an **int** value. Note that not all implementations allow this option to be set.

SO_SNDTIMEO Sets the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it returns with a partial count or with errno set to [EAGAIN] ore [EWOULDBLOCK] if no data were sent. The default for this option is zero, which indicates that a send operation will not time out. This option stores a **timeval** structure. Note that not all implementations allow this option to be set.

For boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.

Options at other protocol levels vary in format and name.

**RETURN VALUE**

Upon successful completion, *setsockopt*( ) returns 0. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *setsockopt*( ) function will fail if:

[EBADF] The *socket* argument is not a valid file descriptor.

[EDOM] The send and receive timeout values are too big to fit into the timeout fields in the socket structure.

[EFAULT] The *option_value* parameter can not be accessed or written.

[EINVAL] The specified option is invalid at the specified socket level or the socket has been shut down.

[EISCONN] The socket is already connected, and a specified option can not be set while the socket is connected.

[ENOPROTOOPT] The option is not supported by the protocol.

[ENOTSOCK] The *socket* argument does not refer to a socket.

The *setsockopt*( ) function may fail if:

[ENOMEM] There was insufficient memory available for the operation to complete.

[ENOBUFS] Insufficient resources are available in the system to complete the call.

[ENOSR] There were insufficient STREAMS resources available for the operation to complete.

**APPLICATION USAGE**

The *setsockopt*( ) function provides an application program with the means to control socket behaviour. An application program can use *setsockopt*( ) to allocate buffer space, control timeouts, or permit socket data broadcasts. The **<sys/socket.h>** header defines the socket-level options available to *setsockopt*( ).

Options may exist at multiple protocol levels. The SO_ options are always present at the uppermost socket level.

**SEE ALSO**

*bind*( ), *endprotoent*( ), *getsockopt*( ), *socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

shutdown — shut down socket send and receive operations

**SYNOPSIS**

```
#include <sys/socket.h>

int shutdown(int socket, int how);
```

**DESCRIPTION**

*socket* Specifies the file descriptor of the socket.

*how* Specifies the type of shutdown. The values are as follows:

| | |
|---|---|
| SHUT_RD | Disables further receive operations. |
| SHUT_WR | Disables further send operations. |
| SHUT_RDWR | Disables further send and receive operations. |

The *shutdown*( ) function disables subsequent send and/or receive operations on a socket, depending on the value of the *how* argument.

**RETURN VALUE**

Upon successful completion, *shutdown*( ) returns 0. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *shutdown*( ) function will fail if:

| | |
|---|---|
| [EBADF] | The *socket* argument is not a valid file descriptor. |
| [EINVAL] | The *how* argument is invalid. |
| [ENOTCONN] | The socket is not connected. |
| [ENOTSOCK] | The *socket* argument does not refer to a socket. |

The *shutdown*( ) function may fail if:

| | |
|---|---|
| [ENOBUFS] | Insufficient resources were available in the system to perform the operation. |
| [ENOSR] | There were insufficient STREAMS resources available for the operation to complete. |

**SEE ALSO**

*getsockopt*( ), *read*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendto*( ), *setsockopt*( ), *socket*( ), *write*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

socket — create an endpoint for communication

**SYNOPSIS**

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

**DESCRIPTION**

The *socket*( ) function creates an unbound socket in a communications domain, and returns a file descriptor that can be used in later function calls that operate on sockets.

The function takes the following arguments:

*domain* Specifies the communications domain in which a socket is to be created.

*type* Specifies the type of socket to be created.

*protocol* Specifies a particular protocol to be used with the socket. Specifying a *protocol* of 0 causes *socket*( ) to use an unspecified default protocol appropriate for the requested socket type.

The *domain* argument specifies the address family used in the communications domain. The address families supported by the system are implementation-dependent.

Symbolic constants that can be used for the domain argument are defined in the **<sys/socket.h>** header.

The *type* argument specifies the socket type, which determines the semantics of communication over the socket. The socket types supported by the system are implementation-dependent. Possible socket types include:

SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.

SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.

SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode transmission path for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

If the *protocol* argument is non-zero, it must specify a protocol that is supported by the address family. The protocols supported by the system are implementation-dependent.

The process may need to have appropriate privileges to use the *socket*( ) function or to create some sockets.

**RETURN VALUE**

Upon successful completion, *socket*( ) returns a nonnegative integer, the socket file descriptor. Otherwise a value of –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *socket*( ) function will fail if:

[EAFNOSUPPORT] The implementation does not support the specified address family.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] No more file descriptors are available for the system.

[EPROTONOSUPPORT]
The protocol is not supported by the address family, or the protocol is not supported by the implementation.

[EPROTOTYPE] The socket type is not supported by the protocol.

The *socket*( ) function may fail if:

[EACCES] The process does not have appropriate privileges.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

[ENOSR] There were insufficient STREAMS resources available for the operation to complete.

**APPLICATION USAGE**

The documentation for specific address families specify which protocols each address family supports. The documentation for specific protocols specify which socket types each protocol supports.

The application can determine if an address family is supported by trying to create a socket with *domain* set to the protocol in question.

**SEE ALSO**

*accept*( ), *bind*( ), *connect*( ), *getsockname*( ), *getsockopt*( ), *listen*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *send*( ), *sendmsg*( ), *setsockopt*( ), *shutdown*( ), *socketpair*( ), **<netinet/in.h>**, **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

socketpair — create a pair of connected sockets

**SYNOPSIS**

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol,
    int socket_vector[2]);
```

**DESCRIPTION**

The *socketpair*( ) function creates an unbound pair of connected sockets in a specified *domain*, of a specified *type*, under the protocol optionally specified by the *protocol* argument. The two sockets are identical. The file descriptors used in referencing the created sockets are returned in *socket_vector*[0] and *socket_vector*[1].

*domain* Specifies the communications domain in which the sockets are to be created.

*type* Specifies the type of sockets to be created.

*protocol* Specifies a particular protocol to be used with the sockets. Specifying a *protocol* of 0 causes *socketpair*( ) to use an unspecified default protocol appropriate for the requested socket type.

*socket_vector* Specifies a 2-integer array to hold the file descriptors of the created socket pair.

The *type* argument specifies the socket type, which determines the semantics of communications over the socket. The socket types supported by the system are implementation-dependent. Possible socket types include:

SOCK_STREAM Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.

SOCK_DGRAM Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.

SOCK_SEQPACKET Provides sequenced, reliable, bidirectional, connection-mode transmission path for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

If the *protocol* argument is non-zero, it must specify a protocol that is supported by the address family. The protocols supported by the system are implementation-dependent.

The process may need to have appropriate privileges to use the *socketpair*( ) function or to create some sockets.

**RETURN VALUE**

Upon successful completion, this function returns 0. Otherwise, –1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *socketpair*( ) function will fail if:

[EAFNOSUPPORT] The implementation does not support the specified address family.

[EMFILE] No more file descriptors are available for this process.

[ENFILE] No more file descriptors are available for the system.

[EOPNOTSUPP] The specified protocol does not permit creation of socket pairs.

[EPROTONOSUPPORT]
The protocol is not supported by the address family, or the protocol is not supported by the implementation.

[EPROTOTYPE] The socket type is not supported by the protocol.

The *socketpair*( ) function may fail if:

[EACCES] The process does not have appropriate privileges.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

[ENOSR] There were insufficient STREAMS resources available for the operation to complete.

**APPLICATION USAGE**

The documentation for specific address families specifies which protocols each address family supports. The documentation for specific protocols specifies which socket types each protocol supports.

The *socketpair*( ) function is used primarily with UNIX domain sockets and need not be supported for other domains.

**SEE ALSO**

*socket*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

write, writev — write on a file

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**DESCRIPTION**

If *fildes* refers to a socket, *write*( ) is equivalent to *send*( ) with no flags set.

**CHANGE HISTORY**

First released in Issue 4.

*Chapter 3*

# *Sockets Headers*

Support for the headers defined in this Chapter is mandatory.

This chapter describes the contents of headers used by the X/Open Sockets functions, macros and external variables.

Headers contain the definition of symbolic constants, common structures, preprocessor macros and defined types. Each function in Chapter 2 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.

**NAME**

fcntl.h — file control options

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**DESCRIPTION**

The **<fcntl.h>** header defines the following additional values for *cmd* used by *fcntl*( ):

| | |
|---|---|
| F_GETOWN | Get process or process group ID to receive SIGURG signals. |
| F_SETOWN | Set process or process group ID to receive SIGURG signals. |

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

net/if.h - sockets local interfaces

**SYNOPSIS**

```
#include <net/if.h>
```

**DESCRIPTION**

The **<net/if.h>** header defines the *if_nameindex* structure that includes at least the following members:

| Member | Type | Value |
|---|---|---|
| if_index | unsigned int | numeric index of the interface |
| if_name | char * | null-terminated name of the interface |

The **<net/if.h>** header defines the following macro for the length of a buffer containing an interface name (including the terminating NULL character):

```
IF_NAMESIZE    interface name length
```

The following are declared as functions, and may also be defined as macros:

```
unsigned int    if_nametoindex(const char *ifname);
char           *if_indextoname(unsigned int ifindex, char *ifname);
struct if_nameindex
               *if_nameindex(void);
void            if_freenameindex
                    (struct if_nameindex *ptr);
```

**SEE ALSO**

*if_freenameindex*( ), *if_indextoname*( ), *if_nameindex*( ), *if_nametoindex*( ).

**CHANGE HISTORY**

First released in Issue 5.2.

**NAME**

sys/socket.h — Main Sockets Header

**SYNOPSIS**

```
#include <sys/socket.h>
```

**DESCRIPTION**

**<sys/socket.h>** makes available a type, **socklen_t**, which is an opaque integral type of length of at least 32 bits[3].

The **<sys/socket.h>** header defines the unsigned integral type **sa_family_t**.

The **<sys/socket.h>** header defines the **sockaddr** structure that includes at least the following members:

| | | |
|---|---|---|
| `sa_family_t` | `sa_family` | address family |
| `char` | `sa_data[]` | socket address (variable-length data) |

The **<sys/socket.h>** header defines the **sockaddr_storage** structure. This structure must be:

- Large enough to accommodate all supported protocol-specific address structures
- aligned at an appropriate boundary so that pointers to it can be cast as pointers to protocol-specific address structures and used to access the fields of those structures without alignment problems.

The **sockaddr_storage** structure contains field **ss_family** which is of type **sa_family_t**. When a **sockaddr_storage** structure is cast as a **sockaddr** structure, the **ss_family** field of the **sockaddr_storage** structure maps onto the **sa_family**field**of**the **sockaddr** structure. When a **sockaddr_storage** structure is cast as a protocol-specific address structure, the **ss_family** field maps onto a field of that structure that is of type **sa_family_t** and that identifies the protocol's address family.

**Note:** Like all notes in this document, this note is non-normative. The **sockaddr_storage** structure solves the problem of declaring storage for automatic variables which is large enough and aligned enough for storing socket address data structure of any family. For example, code with a file descriptor and without the context of the address family can pass a pointer to a variable of this type where a pointer to a socket address structure is expected in calls such as *getpeername*( ) and determine the address family by accessing the received content after the call.

An example implementation design of such a data structure would be as follows:

```
/*
 * Desired design of maximum size and alignment
 */
#define _SS_MAXSIZE 128 /* Implementation specific max size */
#define _SS_ALIGNSIZE (sizeof (int64_t))
                /* Implementation specific desired alignment */
/*
 * Definitions used for sockaddr_storage structure paddings design.
 */
#define _SS_PAD1SIZE (_SS_ALIGNSIZE - sizeof (sa_family_t))
#define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof (sa_family_t)+
                                _SS_PAD1SIZE + _SS_ALIGNSIZE))
```

3. To forestall portability problems, it is recommended that applications should not use values larger than $2^{32} - 1$.

```
struct sockaddr_storage {
    sa_family_t  ss_family;     /* address family */
    /* Following fields are implementation specific */
    char _ss_pad1[_SS_PAD1SIZE];
             /* 6 byte pad, this is to make implementation */
             /* specific pad up to alignment field that */
             /* follows explicit in the data structure */
    int64_t _ss_align;     /* field to force desired structure */
                           /* storage alignment */
    char _ss_pad2[_SS_PAD2SIZE];
             /* 112 byte pad to achieve desired size, */
             /* _SS_MAXSIZE value minus size of ss_family */
             /* __ss_pad1, __ss_align fields is 112 */
};
```

The above example implementation illustrates a data structure which will align on a 64-bit boundary. An implementation-specific field "_ss_align" along "_ss_pad1" is used to force a 64-bit alignment which covers proper alignment good enough for needs of sockaddr_in6 (IPv6), sockaddr_in (IPv4) address data structures. The size of padding fields _ss_pad1 depends on the chosen alignment boundary. The size of padding field _ss_pad2 depends on the value of overall size chosen for the total size of the structure. This size and alignment are represented in the above example by implementation specific (not required) constants _SS_MAXSIZE (chosen value 128) and _SS_ALIGNMENT (with chosen value 8). Constants _SS_PAD1SIZE (derived value 6) and _SS_PAD2SIZE (derived value 112) are also for illustration and not required. The implementation specific definitions and structure field names above start with an underscore to denote implementation private namespace. Portable code is not expected to access or reference those fields or constants.

The **<sys/socket.h>** header defines the **msghdr** structure that includes at least the following members:

| | | |
|---|---|---|
| `void` | `*msg_name` | optional address |
| `socklen_t` | `msg_namelen` | size of address |
| `struct iovec` | `*msg_iov` | scatter/gather array |
| `int` | `msg_iovlen` | members in msg_iov |
| `void` | `*msg_control` | ancillary data, see below |
| `socklen_t` | `msg_controllen` | ancillary data buffer len |
| `int` | `msg_flags` | flags on received message |

The iovec structure is defined through typedef as described in **<sys/uio.h>**.

The **<sys/socket.h>** header defines the **cmsghdr** structure that includes at least the following members:

| | | |
|---|---|---|
| `socklen_t` | `cmsg_len` | data byte count, including the cmsghdr |
| `int` | `cmsg_level` | originating protocol |
| `int` | `cmsg_type` | protocol-specific type |

Ancillary data consists of a sequence of pairs, each consisting of a **cmsghdr** structure followed by a data array. The data array contains the ancillary data message, and the **cmsghdr** structure contains descriptive information that allows an application to correctly parse the data.

The values for **cmsg_level** will be legal values for the level argument to the *getsockopt*( ) and *setsockopt*( ) functions. The system documentation should specify the **cmsg_type** definitions for the supported protocols.

Ancillary data is also possible at the socket level. The **<sys/socket.h>** header defines the following macro for use as the **cmsg_type** value when **cmsg_level** is SOL_SOCKET:

SCM_RIGHTS Indicates that the data array contains the access rights to be sent or received.

The **<sys/socket.h>** header defines the following macros to gain access to the data arrays in the ancillary data associated with a message header:

CMSG_DATA(*cmsg*) If the argument is a pointer to a **cmsghdr** structure, this macro returns an unsigned character pointer to the data array associated with the **cmsghdr** structure.

CMSG_NXTHDR(*mhdr,cmsg*)
If the first argument is a pointer to a **msghdr** structure and the second argument is a pointer to a **cmsghdr** structure in the ancillary data, pointed to by the **msg_control** field of that **msghdr** structure, this macro returns a pointer to the next **cmsghdr** structure, or a null pointer if this structure is the last **cmsghdr** in the ancillary data.

CMSG_FIRSTHDR(*mhdr*)
If the argument is a pointer to a **msghdr** structure, this macro returns a pointer to the first **cmsghdr** structure in the ancillary data associated with this **msghdr** structure, or a null pointer if there is no ancillary data associated with the **msghdr** structure.

The **<sys/socket.h>** header defines the **linger** structure that includes at least the following members:

| | | |
|---|---|---|
| `int` | `l_onoff` | indicates whether linger option is enabled |
| `int` | `l_linger` | linger time, in seconds |

The **<sys/socket.h>** header defines the following macros, with distinct integral values:

| | |
|---|---|
| SOCK_DGRAM | Datagram socket |
| SOCK_STREAM | Byte-stream socket |
| SOCK_SEQPACKET | Sequenced-packet socket |

The **<sys/socket.h>** header defines the following macro for use as the *level* argument of *setsockopt*( ) and *getsockopt*( ).

SOL_SOCKET Options to be accessed at socket level, not protocol level.

The **<sys/socket.h>** header defines the following macros, with distinct integral values, for use as the *option_name* argument in *getsockopt*( ) or *setsockopt*( ) calls:

| | |
|---|---|
| SO_ACCEPTCONN | Socket is accepting connections. |
| SO_BROADCAST | Transmission of broadcast messages is supported. |
| SO_DEBUG | Debugging information is being recorded. |
| SO_DONTROUTE | bypass normal routing |
| SO_ERROR | Socket error status. |
| SO_KEEPALIVE | Connections are kept alive with periodic messages. |
| SO_LINGER | Socket lingers on close. |
| SO_OOBINLINE | Out-of-band data is transmitted in line. |
| SO_RCVBUF | Receive buffer size. |
| SO_RCVLOWAT | receive "low water mark" |
| SO_RCVTIMEO | receive timeout |
| SO_REUSEADDR | Reuse of local addresses is supported. |
| SO_SNDBUF | Send buffer size. |
| SO_SNDLOWAT | send "low water mark" |
| SO_SNDTIMEO | send timeout |

| | |
|---|---|
| SO_TYPE | Socket type. |

The **<sys/socket.h>** header defines the following macros, with distinct integral values, for use as the valid values for the **msg_flags** field in the **msghdr** structure, or the flags parameter in *recvfrom*( ), *recvmsg*( ), *sendto*( ) or *sendmsg*( ) calls:

| | |
|---|---|
| MSG_CTRUNC | Control data truncated. |
| MSG_DONTROUTE | Send without using routing tables. |
| MSG_EOR | Terminates a record (if supported by the protocol). |
| MSG_OOB | Out-of-band data. |
| MSG_PEEK | Leave received data in queue. |
| MSG_TRUNC | Normal data truncated. |
| MSG_WAITALL | Wait for complete message. |

The **<sys/socket.h>** header defines the following macros, with distinct integral values:

| | |
|---|---|
| AF_UNIX | UNIX domain sockets |
| AF_UNSPEC | Unspecified |
| AF_INET | Internet domain sockets for use with IPv4 addresses |
| AF_INET6 | Internet domain sockets for use with IPv6 addresses |

The **<sys/socket.h>** header defines the following macros, with distinct integral values:

| | |
|---|---|
| SHUT_RD | Disables further receive operations. |
| SHUT_WR | Disables further send operations. |
| SHUT_RDWR | Disables further send and receive operations. |

The following are declared as functions, and may also be defined as macros:

```
int     accept(int socket, struct sockaddr *address,
            socklen_t *address_len);
int     bind(int socket, const struct sockaddr *address,
            socklen_t address_len);
int     connect(int socket, const struct sockaddr *address,
            socklen_t address_len);
int     getpeername(int socket, struct sockaddr *address,
            socklen_t *address_len);
int     getsockname(int socket, struct sockaddr *address,
            socklen_t *address_len);
int     getsockopt(int socket, int level, int option_name,
            void *option_value, socklen_t *option_len);
int     listen(int socket, int backlog);
ssize_t recv(int socket, void *buffer, size_t length, int flags);
ssize_t recvfrom(int socket, void *buffer, size_t length,
            int flags, struct sockaddr *address, socklen_t *address_len);
ssize_t recvmsg(int socket, struct msghdr *message, int flags);
ssize_t send(int socket, const void *message, size_t length, int flags);
ssize_t sendmsg(int socket, const struct msghdr *message, int flags);
ssize_t sendto(int socket, const void *message, size_t length, int flags,
            const struct sockaddr *dest_addr, socklen_t dest_len);
int     setsockopt(int socket, int level, int option_name,
            const void *option_value, socklen_t option_len);
int     shutdown(int socket, int how);
int     socket(int domain, int type, int protocol);
int     socketpair(int domain, int type, int protocol,
            int socket_vector[2]);
```

Inclusion of **<sys/socket.h>** may also make visible all symbols from **<sys/uio.h>**.

**SEE ALSO**

*accept*( ), *bind*( ), *connect*( ), *getpeername*( ), *getsockname*( ), *getsockopt*( ), *listen*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *send*( ), *sendmsg*( ), *sendto*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), *socketpair*( ), **<sys/uio.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

sys/stat.h — data returned by the *stat*( ) function.

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to Sockets.

**DESCRIPTION**

The following additional symbolic name for the value of **st_mode** is defined:

File type:

S_IFMT type of file
S_IFSOCK socket

The following macro will test whether a file is of the specified type. The value *m* supplied to the macro is the value of **st_mode** from a **stat** structure. The macro evaluates to a non-zero value if the test is true, 0 if the test is false.

S_ISSOCK(*m*) test for a socket

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

sys/uio.h — definitions for scatter/gather I/O

**Note:** The **XSH** specification contains the basic definition of this interface.

**DESCRIPTION**

This specification uses the **iovec** structure defined in **<sys/uio.h>** as definitions for scatter/gather I/O.

*Chapter 4*

# *IP Address Resolution Interfaces*

Support for the IP Address Resolution interfaces defined in this Chapter is mandatory.

**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).

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.

**NAME**

endhostent, freehostent, gethostbyaddr, gethostbyname, gethostent, getipnodebyaddr, getipnodebyname, sethostent — network host database functions

**SYNOPSIS**

```
#include <netdb.h>

struct hostent *gethostent(void);

void sethostent(int stayopen);

void endhostent(void);

struct hostent *getipnodebyaddr
    (const void *addr, socklen_t len, int type, int *error_num);

struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);

struct hostent *getipnodebyname
    (const char *name, int type, int flags, int *error_num);

struct hostent *gethostbyname(const char *name);

void freehostent(struct hostent *ptr);
```

**DESCRIPTION**

These functions enable applications to retrieve information about hosts. This information is considered to be stored in a database that can be accessed sequentially or randomly. Implementation of this database is unspecified.

**Note:** In many cases it will be implemented by the Domain Name System, see referenced documents **RFC 1034**, **RFC 1035** and **RFC 1886**.

Entries are returned in **hostent** structures.

The *sethostent*( ) function opens a connection to the database and sets the next entry for retrieval to the first entry in the database. If the *stayopen* argument is non-zero, the connection will not be closed by a call to *gethostent*( ), *getipnodebyname*( ), *gethostbyname*( ), *getipnodebyaddr*( ), or *gethostbyaddr*( ).

The *gethostent*( ) function reads the next entry in the database, opening a connection to the database if necessary.

The *endhostent*( ) function closes the connection to the database.

The *getipnodebyaddr*( ) function returns the entry containing addresses of address family *type* for the host with address *addr*, opening a connection to the database if necessary. Argument *len* contains the length of the address pointed to by *addr*. If an error occurs, the appropriate error code is returned in *error_num*. Function *getipnodebyaddr*( ) is thread-safe.

The *gethostbyaddr*( ) function is a legacy function whose use is deprecated in favour of *getipnodebyaddr*( ). It returns an entry containing addresses of address family *type* for the host with address *addr*. *len* contains the length of the address pointed to by *addr*. It need not be thread-safe.

The *addr* argument of *getipnodebyaddr*( ) or *gethostbyaddr*( ) must be an **in_addr** structure when *type* is AF_INET, and must be an **in6_addr** structure when *type* is AF_INET6. It contains a binary format (that is, not null-terminated) address in network byte order. The *gethostbyaddr*( ) function is not guaranteed to return addresses of address families other than AF_INET, even when such addresses exist in the database.

If *gethostbyaddr*( ) or *getipnodebyaddr*( ) returns a record then its **h_addrtype** field is the same as the *type* argument that was passed to the function, and its **h_addr_list** field lists a single address that is a copy of the *addr* argument that was passed to the function. If *type* is AF_INET6 and *addr* is an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address then the **h_name** and **h_aliases** fields are those that would have been returned for address family AF_INET and address equal to the last four bytes of *addr*.

If *getipnodebyaddr*( ) or *gethostbyaddr*( ) is called with *addr* containing the IPv6 unspecified address (all bytes zero) then no query is performed and the function fails with error [HOST_NOT_FOUND].

The *getipnodebyname*( ) function returns the entry containing addresses of address family *type* for the host with name *name*, opening a connection to the database if necessary. Argument *flags* affects what information is returned. If an error occurs, the appropriate error code is returned in *error_num*. Function *getipnodebyname*( ) is thread-safe.

The *gethostbyname*( ) function is a legacy function whose use is deprecated in favour of *getipnodebyname*( ). It returns an entry containing addresses of address family AF_INET for the host with name *name*. It need not be thread-safe.

The *name* argument of *getipnodebyname*( ) can be either a node name or a numeric address string. For IPv4 a numeric address string will be in the dotted-decimal notation described on the man page for *inet_addr*( ). For IPv6 a numeric address string will be in one of the standard IPv6 text forms described on the man page for *inet_pton*( ). The *name* argument of *gethostbyname*( ) can be a node name; the behavior of *gethostbyname*( ) when passed a numeric address string is unspecified.

If *name* is a dotted-decimal IPv4 address and *af* equals AF_INET, or name is an IPv6 hex address and *af* equals AF_INET6, the members of the returned hostent structure are as follows:

| | |
|---|---|
| *h_name* | points to a copy of the *name* argument |
| *h_aliases* | is a NULL pointer |
| *h_addrtype* | is a copy of the *type* argument |
| *h_length* | is either 4 (for AF_INET) or 16 (for AF_INET6) |
| *h_addr_list[0]* | is a pointer to the 4-byte or 16-byte binary address |
| *h_addr_list[1]* | is a NULL pointer |

If *name* is a dotted-decimal IPv4 address and *af* equals AF_INET6 and AI_V4MAPPED is set in *flags*, an IPv4-mapped IPv6 address is returned, and:

| | |
|---|---|
| *h_name* | points to an IPv6 hex address containing the IPv4-mapped IPv6 address |
| *h_aliases* | is a NULL pointer |
| *h_addrtype* | is AF_INET6 |
| *h_length* | is 16 |
| *h_addr_list[0]* | is a pointer to the 16-byte binary address |
| *h_addr_list[1]* | is a NULL pointer |

If *name* is a dotted-decimal IPv4 address and *af* equals AF_INET6 and AI_V4MAPPED is not set, then NULL is returned with error [HOST_NOT_FOUND].

It is an error when *name* is an IPv6 hex address and *af* equals AF_INET. The function's return value is a NULL pointer with error [HOST_NOT_FOUND].

If *name* is not a numeric address string and is an alias for a valid host name then *getipnodebyname*( ) or *gethostbyname*( ) returns information about the host name to which the alias refers, and *name* is included in the list of aliases returned.

If *name* is a node name then operation of the *getipnodebyname*( ) function is modified by the value of the *flags* argument, as follows:

- If *flags* is 0 and *type* is AF_INET, then a query is made for IPv4 addresses. If it is successful, the IPv4 addresses are returned and the **h_length** member of the **hostent** structure will be 4. Otherwise, the function returns a NULL pointer.
- If *flags* is 0 and if *type* is AF_INET6, then a query is made for IPv6 addresses. If it is successful, the IPv6 addresses are returned and the **h_length** member of the hostent structure will be 16. If unsuccessful, the function returns a NULL pointer.
- If the AI_V4MAPPED flag is set and *type* is AF_INET6, then a query is made for IPv6 addresses. If it is successful, the IPv6 addresses are returned, and no query is made for IPv4 addresses. If it is not successful, a query is made for IPv4 addresses and any found are returned as IPv4-mapped IPv6 addresses. **h_length** will be 16 in either case of addresses being returned. The AI_V4MAPPED flag is ignored unless *type* is AF_INET6.
- If the AI_ALL and AI_V4MAPPED flags are both set and *type* is AF_INET6, then a query is made for IPv6 addresses, and any found are returned. Another query is then made for IPv4 addresses, and any found are returned as IPv4-mapped IPv6 addresses, and **h_length** is 16. Only if both queries fail does the function return a NULL pointer. This flag is ignored unless type is AF_INET6.
- The AI_ADDRCONFIG flag specifies that a query for IPv6 addresses should be made only if the node has at least one IPv6 source address configured, and that a query for IPv4 addresses should be made only if the node has at least one IPv4 source address configured.
- If the AI_V4MAPPED and AI_ADDRCONFIG flags are both set and *type* is AF_INET6, then
  - — If the node has at least one IPv6 source address configured, a query is made for IPv6 addresses
  - — If it is successful, the IPv6 addresses are returned and no query is made for IPv4 addresses
  - — If the node has no IPv6 source address configured, or if the query for IPv6 addresses is not successful, then if the node has at least one IPv4 source address configured, a query is made for IPv4 addresses and any found are returned as IPv4-mapped IPv6 addresses.

  **h_length** will be 16 in either case of addresses being returned.
- Macro AI_DEFAULT is defined as the logical OR of AI_V4MAPPED and AI_ADDRCONFIG.

  **Note:** It is intended that setting *flags* to AI_DEFAULT will be appropriate for most applications.

The *freehostent*( ) function frees the memory occupied by the hostent structure pointed to by **hostent** and any structures pointed to from that structure, provided that **hostent** was obtained by a call to *getipnodebyaddr*( ) or *getipnodebyname*( ). Applications must not call *freehostent*( ) except to pass it a pointer that was obtained from *getipnodebyaddr*( ) or *getipnodebyname*( ).

**RETURN VALUE**

On successful completion, *getipnodebyaddr*( ), *gethostbyaddr*( ), *getipnodebyname*( ), *gethostbyname*( ) and *gethostent*( ) return a pointer to a **hostent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found.

On unsuccessful completion, *getipnodebyaddr*( ) and *getipnodebyname*( ) will set their *error_num* argument to indicate the error, while *gethostbyaddr*( ) and *gethostbyname*( ) will set *h_errno* to indicate it.

**ERRORS**

No errors are defined for *endhostent*( ), *gethostent*( ) and *sethostent*( ).

The *getipnodebyaddr*( ), *getipnodebyname*( ), *gethostbyaddr*( ) and *gethostbyname*( ) functions will fail in the following cases. Functions *getipnodebyaddr*( ) and *getipnodebyname*( ) will return the value shown in the list below in *error_num;* functions *gethostbyaddr*( ) and *gethostbyname*( ) will set *h_errno* to that value. Any changes to *errno* are unspecified.

[HOST_NOT_FOUND]
No such host is known.

[NO_DATA] The server recognised the request and the name but no address is available. Another type of request to the name server for the domain might return an answer.

[NO_RECOVERY] An unexpected server failure occurred which can not be recovered.

[TRY_AGAIN] A temporary and possibly transient error occurred, such as a failure of a server to respond.

**APPLICATION USAGE**

The hostent structure returned by *getipnodebyaddr*( ) and *getipnodebyname*( ), and any structures pointed to from those structures, are dynamically allocated. Applications should call *freehostent*( ) to free the memory used by these structures when they are no longer needed.

The *gethostent*( ), *gethostbyaddr*( ), and *gethostbyname*( ) functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions. Applications must not call *freehostent*( ) for this area.

**SEE ALSO**

*endservent*( ), *htonl*( ), *inet_addr*( ), **<netdb.h>**.

**CHANGE HISTORY**

First released in Issue 4.

Specifications of *getipnodebyaddr*( ). *getipnodebyname*( ) and *freehostent*( ) were added in Issue 5.2.

**NAME**

endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

**SYNOPSIS**

```
#include <netdb.h>

void endnetent(void);

struct netent *getnetbyaddr(uint32_t net, int type);

struct netent *getnetbyname(const char *name);

struct netent *getnetent(void);

void setnetent(int stayopen);
```

**DESCRIPTION**

The *getnetbyaddr*( ), *getnetbyname*( ) and *getnetent*( ), functions each return a pointer to a **netent** structure, the members of which contain the fields of an entry in the network database.

The *getnetent*( ) function reads the next entry of the database, opening a connection to the database if necessary.

The *getnetbyaddr*( ) function searches the database from the beginning, and finds the first entry for which the address family specified by *type* matches the **n_addrtype** member and the network number *net* matches the **n_net** member, opening a connection to the database if necessary. The *net* argument is the network number in host byte order.

The *getnetbyname*( ) function searches the database from the beginning and finds the first entry for which the network name specified by *name* matches the **n_name** member, opening a connection to the database if necessary.

The *setnetent*( ) function opens and rewinds the database. If the *stayopen* argument is non-zero, the connection to the net database will not be closed after each call to *getnetent*( ) (either directly, or indirectly through one of the other *getnet**( ) functions).

The *endnetent*( ) function closes the database.

**RETURN VALUE**

On successful completion, *getnetbyaddr*( ), *getnetbyname*( ) and *getnetent*( ), return a pointer to a **netent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

**ERRORS**

No errors are defined.

**APPLICATION USAGE**

The *getnetbyaddr*( ), *getnetbyname*( ) and *getnetent*( ), functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

**SEE ALSO**

**<netdb.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

endprotoent, getprotobynumber, getprotobyname, getprotoent, setprotoent — network protocol database functions

**SYNOPSIS**

```
#include <netdb.h>

void endprotoent(void);

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void);

void setprotoent(int stayopen);
```

**DESCRIPTION**

The *getprotobyname*( ), *getprotobynumber*( ) and *getprotoent*( ), functions each return a pointer to a **protoent** structure, the members of which contain the fields of an entry in the network protocol database.

The *getprotoent*( ) function reads the next entry of the database, opening a connection to the database if necessary.

The *getprotobyname*( ) function searches the database from the beginning and finds the first entry for which the protocol name specified by *name* matches the **p_name** member, opening a connection to the database if necessary.

The *getprotobynumber*( ) function searches the database from the beginning and finds the first entry for which the protocol number specified by *number* matches the **p_proto** member, opening a connection to the database if necessary.

The *setprotoent*( ) function opens a connection to the database, and sets the next entry to the first entry. If the *stayopen* argument is non-zero, the connection to the network protocol database will not be closed after each call to *getprotoent*( ) (either directly, or indirectly through one of the other *getproto**( ) functions).

The *endprotoent*( ) function closes the connection to the database.

**RETURN VALUES**

On successful completion, *getprotobyname*( ), *getprotobynumber*( ) and *getprotoent*( ) functions return a pointer to a **protoent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

**ERRORS**

No errors are defined.

**APPLICATION USAGE**

The *getprotobyname*( ), *getprotobynumber*( ) and *getprotoent*( ) functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

**SEE ALSO**

**<netdb.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

endservent, getservbyport, getservbyname, getservent, setservent — network services database functions

**SYNOPSIS**

```
#include <netdb.h>

void endservent(void);

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

struct servent *getservent(void);

void setservent(int stayopen);
```

**DESCRIPTION**

The *getservbyname*( ), *getservbyport*( ) and *getservent*( ) functions each return a pointer to a **servent** structure, the members of which contain the fields of an entry in the network services database.

The *getservent*( ) function reads the next entry of the database, opening a connection to the database if necessary.

The *getservbyname*( ) function searches the database from the beginning and finds the first entry for which the service name specified by *name* matches the **s_name** member and the protocol name specified by *proto* matches the **s_proto** member, opening a connection to the database if necessary. If *proto* is a null pointer, any value of the **s_proto** member will be matched.

The *getservbyport*( ) function searches the database from the beginning and finds the first entry for which the port specified by *port* matches the **s_port** member and the protocol name specified by *proto* matches the **s_proto** member, opening a connection to the database if necessary. If *proto* is a null pointer, any value of the **s_proto** member will be matched. The *port* argument must be in network byte order.

The *setservent*( ) function opens a connection to the database, and sets the next entry to the first entry. If the *stayopen* argument is non-zero, the net database will not be closed after each call to the *getservent*( ) function (either directly, or indirectly through one of the other *getserv**( ) functions).

The *endservent*( ) function closes the database.

**RETURN VALUES**

On successful completion, *getservbyname*( ), *getservbyport*( ) and *getservent*( ) return a pointer to a **servent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

**ERRORS**

No errors are defined.

**APPLICATION USAGE**

The *port* argument of *getservbyport*( ) need not be compatible with the port values of all address families.

The *getservent*( ), *getservbyname*( ) and *getservbyport*( ) functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

**SEE ALSO**

*endhostent*( ), *endprotoent*( ), *htonl*( ), *inet_addr*( ), **<netdb.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

gai_strerror — address and name information error description

**SYNOPSIS**

```
#include <netdb.h>

char *gai_strerror(int ecode);
```

**DESCRIPTION**

Function *gai_strerror*( ) returns a text string describing an error that is listed in **Address Information Structure** on page 101.

When argument *ecode* is one of the values listed in **Address Information Structure** on page 101, the function return value points to a string describing the error. If the argument is not one of those values, the function returns a pointer to a string whose contents indicate an unknown error.

**SEE ALSO**

*getaddrinfo*( ), **<netdb.h>**.

**CHANGE HISTORY**

First released in Issue 5.2.

**NAME**

getaddrinfo, freeaddrinfo — get address information[4]

**SYNOPSIS**

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char  *nodename, const char *servname,
    const struct addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);
```

**DESCRIPTION**

The *getaddrinfo*( ) function translates the name of a service location (for example, a host name) and/or a service name and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

The *nodename* and *servname* arguments are either null pointers or pointers to null-terminated strings. One or both of these two arguments must be a non-null pointer.

The format of a valid name depends on the protocol family or families. If a specific family is not given and the name could be interpreted as valid within multiple supported families, the implementation will attempt to resolve the name in all supported families and, in absence of errors, one or more successful results will be returned

If the *nodename* argument is not null, it can be a descriptive name or can be an address string. If the specified address family is AF_INET, AF_INET6 or AF_UNSPEC, valid descriptive names include host names as specified in . If the specified address family is AF_INET or AF_UNSPEC, address strings using Internet standard dot notation as specified on the *inet_addr*( ) man page are valid.

If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described on the *inet_pton*( ) man page are valid.

If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the requested service location is local to the caller.

If *servname* is null, the call returns network-level addresses for the specified *nodename*. If *servname* is not null, it is a null-terminated character string identifying the requested service. This can be either a descriptive name or a numeric representation suitable for use with the address family or families. If the specified address family is AF_INET, AF_INET6 or AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

If the argument *hints* is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In this hints structure every member other than **ai_flags**, **ai_family**, **ai_socktype** and **ai_protocol** must be zero or a null pointer. A value of AF_UNSPEC for **ai_family** means that the caller will accept any protocol family. A value of zero for **ai_socktype** means that the caller will accept any socket type. A value of zero for **ai_protocol** means that the caller will accept any protocol. If *hints* is a null pointer, the behavior must be as if it referred to a structure containing the value zero for the **ai_flags**, **ai_socktype** and **ai_protocol** fields, and AF_UNSPEC for the **ai_family** field.

4. Based on functions defined in IEEE P1003.1g/D6.6, Copyright 1997, IEEE. All rights reserved.

**Notes:**

1. If the caller handles only TCP and not UDP, for example, then the **ai_protocol** member of the hints structure should be set to IPPROTO_TCP when *getaddrinfo*( ) is called.
2. If the caller handles only IPv4 and not IPv6, then the **ai_family** member of the hints structure should be set to PF_INET when *getaddrinfo*( ) is called.

The **ai_flags** field to which *hints* parameter points must have the value zero or be the bitwise OR of one or more of the values AI_PASSIVE, AI_CANONNAME and AI_NUMERICHOST.

If the flag AI_PASSIVE is specified, the returned address information must be suitable for use in binding a socket for accepting incoming connections for the specified service. In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the flag AI_PASSIVE is not specified, the returned address information must be suitable for a call to *connect*( ) (for a connection-mode protocol) or for a call to *connect*( ), *sendto*( ) or *sendmsg*( ) (for a connectionless protocol). In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure will be set to the loopback address.

If the flag AI_CANONNAME is specified and the *nodename* argument is not null, the function attempts to determine the canonical name corresponding to *nodename* (for example, if *nodename* is an alias or shorthand notation for a complete name).

If the flag AI_NUMERICHOST is specified then a non-null *nodename* string must be a numeric host address string. Otherwise an error of [EAI_NONAME] is returned. This flag prevents any type of name resolution service (for example, the DNS) from being invoked.

If the flag AI_NUMERICSERV is specified then a non-null *servname* string must be a numeric port string. Otherwise an error [EAI_NONAME] is returned. This flag prevents any type of name resolution service (for example, NIS+) from being invoked.

The **ai_socktype** field to which argument *hints* points specifies the socket type for the service, as defined on the *socket*( ) man page. If a specific socket type is not given (for example, a value of zero) and the service name could be interpreted as valid with multiple supported socket types, the implementation will attempt to resolve the service name for all supported socket types and, in absense of errors, all possible successful results will be returned. A non-zero socket type value will limit the returned information to values with the specified socket type.

If the **ai_family** field to which *hints* points has the value AF_UNSPEC, addresses are returned for use with any protocol family that can be used with the specified *nodename* and/or *servname*. Otherwise, addresses are returned for use only with the specified protocol family. If **ai_family** is not AF_UNSPEC and **ai_protocol** is not zero, then addresses are returned for use only with the specified protocol family and protocol; the value of **ai_protocol** is interpreted as in a call to the *socket*( ) function with the corresponding values of **ai_family** and **ai_protocol**.

The *freeaddrinfo*( ) function frees one or more **addrinfo** structures returned by *getaddrinfo*( ), along with any additional storage associated with those structures. If the **ai_next** field of the structure is not null, the entire list of structures is freed. The *freeaddrinfo*( ) function must support the freeing of arbitrary sublists of an **addrinfo** list originally returned by *getaddrinfo*( ).

Functions *getaddrinfo*( ) and *freeaddrinfo*( ) must be thread-safe.

**RETURN VALUE**

A zero return value for *getaddrinfo*( ) indicates successful completion; a non-zero return value indicates failure.

Upon successful return of *getaddrinfo*( ), the location to which *res* points refers to a linked list of **addrinfo** structures, each of which specifies a socket address and information for use in creating a socket with which to use that socket address. The list must include at least one **addrinfo** structure. The **ai_next** field of each structure contains a pointer to the next structure on the list, or a null pointer if it is the last structure on the list. Each structure on the list includes values for use with a call to the *socket*( ) function, and a socket address for use with the *connect*( ) function or, if the AI_PASSIVE flag was specified, for use with the *bind*( ) function. The fields **ai_family**, **ai_socktype**, and **ai_protocol** are usable as the arguments to the *socket*( ) function to create a socket suitable for use with the returned address. The fields **ai_addr** and **ai_addrlen** are usable as the arguments to the *connect*( ) or *bind*( ) functions with such a socket, according to the AI_PASSIVE flag.

If *nodename* is not null, and if requested by the AI_CANONNAME flag, the **ai_canonname** field of the first returned **addrinfo** structure points to a null-terminated string containing the canonical name corresponding to the input *nodename*; if the canonical name is not available, then **ai_canonname** refers to the argument *nodename* or a string with the same contents. The contents of the **ai_flags** field of the returned structures is undefined.

All fields in socket address structures returned by *getaddrinfo*( ) that are not filled in through an explicit argument (for example, **sin6_flowinfo** and **sin_zero**) must be set to zero.

**Note:** This makes it easier to compare socket address structures.

**ERRORS**

| | |
|---|---|
| [EAI_AGAIN] | The name could not be resolved at this time. Future attempts may succeed. |
| [EAI_BADFLAGS] | The flags parameter had an invalid value. |
| [EAI_FAIL] | A non-recoverable error occurred when attempting to resolve the name. |
| [EAI_FAMILY] | The address family was not recognized. |
| [EAI_MEMORY] | There was a memory allocation failure when trying to allocate storage for the return value. |
| [EAI_NONAME] | The name does not resolve for the supplied parameters.<br>Neither *nodename* nor *servname* were passed. At least one of these must be passed. |
| [EAI_SERVICE] | The service passed was not recognized for the specified socket type. |
| [EAI_SOCKTYPE] | The intended socket type was not recognized. |
| [EAI_SYSTEM] | A system error occurred; the error code can be found in *errno*. |

**SEE ALSO**

*connect*( ), *gethostbyname*( ), *getipnodebyname*( ), *getnameinfo*( ), *getservbyname*( ), *socket*( ). **<netdb.h>**, **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 5.2.

**NAME**

gethostname — get name of current host

**SYNOPSIS**

```
#include <unistd.h>

int gethostname(char *name, socklen_t namelen);
```

**DESCRIPTION**

The *gethostname*( ) function returns the standard host name for the current machine. The *namelen* argument specifies the size of the array pointed to by the *name* argument. The returned name is null-terminated, except that if *namelen* is an insufficient length to hold the host name, then the returned name is truncated and it is unspecified whether the returned name is null-terminated.

Host names are limited to 255 bytes.

**RETURN VALUE**

On successful completion, 0 is returned. Otherwise, –1 is returned.

**ERRORS**

No errors are defined.

**SEE ALSO**

Man-page definitions in the **XSH** specification, for *gethostid*( ) *uname*( ), **<unistd.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

getnameinfo — get name information

**SYNOPSIS**

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
    char *node, socklen_t nodelen, char *service,
    socklen_t servicelen, unsigned int flags);
```

**DESCRIPTION**

The *getnameinfo*( ) translates a socket address to a node name and service location, all of which are defined as with *getaddrinfo*( ).

The argument *sa* points to a socket address structure to be translated.

If the argument node is non-NULL and the argument *nodelen* is nonzero, then the argument *node* points to a buffer able to contain up to *nodelen* characters that will receive the node name as a null-terminated string. If the argument *node* is NULL or the argument *nodelen* is zero, the node name will not be returned. If the node's name cannot be located, the numeric form of the node's address is returned instead of its name.

If the argument *service* is non-NULL and the argument *servicelen* is nonzero, then the argument *service* points to a buffer able to contain up to *servicelen* characters that will receive the service name as a null-terminated string. If the argument *service* is NULL or the argument *servicelen* is zero, the service name will not be returned. If the service's name cannot be located, the numeric form of the service address (for example, its port number) is returned instead of its name.

The arguments *node* and *service* cannot both be NULL.

The *flags* argument is a flag that changes the default actions of the function. By default the fully-qualified domain name (FQDN) for the host is returned, but

- If the flag bit NI_NOFQDN is set, only the nodename portion of the FQDN is returned for local hosts.
- If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address is returned instead of its name, under all circumstances.
- If the flag bit NI_NAMEREQD is set, an error is returned if the host's name cannot be located.
- If the flag bit NI_NUMERICSERV is set, the numeric form of the service address is returned (for example, its port number) instead of its name, under all circumstances.
- If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service (SOCK_DGRAM). The default behavior is to assume that the service is a stream service (SOCK_STREAM).

**Notes:**

1. The two NI_NUMERICxxx flags are required to support the "-n" flag that many commands provide.
2. The NI_DGRAM flag is required for the few AF_INET/AF_INET6 port numbers (for example, 512-514) that represent different services for UDP and TCP.

Function *getnameinfo*( ) must be thread-safe.

**RETURN VALUE**

A zero return value for *getnameinfo*( ) indicates successful completion; a non-zero return value indicates failure.

On successful completion, function *getnameinfo*( ) returns the node and service names, if requested, in the buffers provided. The returned names are always null-terminated strings, and may be truncated if the actual values are longer than can be stored in the buffers provided. If the returned values are to be used as part of any further name resolution (for example, passed to *getaddrinfo*( )), callers must either provide buffers large enough to store any result possible on the system or must check for truncation and handle that case appropriately.

**ERRORS**

[EAI_AGAIN]
The name could not be resolved at this time. Future attempts may succeed.

[EAI_BADFLAGS]
The flags had an invalid value.

[EAI_FAIL]
A non-recoverable error occurred.

[EAI_FAMILY]
The address family was not recognized or the address length was invalid for the specified family.

[EAI_MEMORY]
There was a memory allocation failure.

[EAI_NONAME]
The name does not resolve for the supplied parameters.

NI_NAMEREQD is set and the host's name cannot be located, or both *nodename* and *servname* were null.

[EAI_SYSTEM]
A system error occurred. The error code can be found in errno.

**SEE ALSO**

*getaddrinfo*( ), *getservbyname*( ), *getservbyport*( ), *inet_ntop*( ), *socket*( ), **<netdb.h>**, **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 5.2.

**NAME**

h_errno — error return value for network database operations

**SYNOPSIS**

```
#include <netdb.h>
```

**DESCRIPTION**

Refer to *endhostent*( ). Note that this method of returning errors is used only in connection with legacy functions.

The **<netdb.h>** header provides a declaration of *h_errno* as a modifiable l-value of type **int**.

Applications should obtain the definition of *h_errno* by the inclusion of **<netdb.h>**. The practice of defining *h_errno* in a program as an *extern int h_errno* is obsolescent.

It is unspecified whether *h_errno* is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name *h_errno*, the behaviour is undefined.

**SEE ALSO**

**errno**

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

htonl, htons, ntohl, ntohs — convert values between host and network byte order

**SYNOPSIS**

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);
```

**DESCRIPTION**

These functions convert 16-bit and 32-bit quantities between network byte order and host byte order.

The **uint32_t** and **uint16_t** types are made available by inclusion of **<inttypes.h>** (see referenced document **XSH**).

**RETURN VALUES**

The *htonl*( ) and *htons*( ) functions return the argument value converted from host to network byte order.

The *ntohl*( ) and *ntohs*( ) functions return the argument value converted from network to host byte order.

**ERRORS**

No errors are defined.

**APPLICATION USAGE**

These functions are most often used in conjunction with Internet IPv4 addresses and ports as returned by *gethostent*( ) and *getservent*( ).

On some architectures these functions are defined as macros that expand to the value of their argument.

**SEE ALSO**

*endhostent*( ), *endservent*( ), **<arpa/inet.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof, inet_ntoa — IPv4 address manipulation

**SYNOPSIS**

```
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);

in_addr_t inet_lnaof(struct in_addr in);

struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);

in_addr_t inet_netof(struct in_addr in);

in_addr_t inet_network(const char *cp);

char *inet_ntoa(struct in_addr in);
```

**DESCRIPTION**

Functions *inet_lnaof*( ), *inet_makeaddr*( ), *inet_netof*( ) and *inet_network*( ) are legacy. They should not be used by new applications.

The *inet_addr*( ) function converts the string pointed to by *cp*, in the standard IPv4 dotted decimal notation, to an integer value suitable for use as an Internet address.

The *inet_lnaof*( ) function takes an Internet host address specified by *in* and extracts the local network address part, in host byte order.

The *inet_makeaddr*( ) function takes the Internet network number specified by *net* and the local network address specified by *lna*, both in host byte order, and constructs an Internet address from them.

The *inet_netof*( ) function takes an Internet host address specified by *in* and extracts the network number part, in host byte order.

The *inet_network*( ) function converts the string pointed to by *cp*, in the standard IPv4 dotted decimal notation, to an integer value suitable for use as an Internet network number.

The *inet_ntoa*( ) function converts the Internet host address specified by *in* to a string in the Internet standard dot notation.

All Internet addresses are returned in network order (bytes ordered from left to right).

Values specified using IPv4 dotted decimal notation take one of the following forms:

`a.b.c.d` When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

`a.b.c` When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as `128.net`.host.

`a.b` When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the rightmost three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as `net`.host.

`a` When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in IPv4 dotted decimal notation may be decimal, octal, or hexadecimal, as specified in the ISO C standard (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

**RETURN VALUE**

Upon successful completion, *inet_addr*( ) returns the Internet address. Otherwise, it returns (**in_addr_t**)(−1).

Upon successful completion, *inet_network*( ) returns the converted Internet network number. Otherwise, it returns (**in_addr_t**)(−1).

The *inet_makeaddr*( ) function returns the constructed Internet address.

The *inet_lnaof*( ) function returns the local network address part.

The *inet_netof*( ) function returns the network number.

The *inet_ntoa*( ) function returns a pointer to the network address in Internet-standard dot notation.

**ERRORS**

No errors are defined.

**APPLICATION USAGE**

The return value of *inet_ntoa*( ) may point to static data that may be overwritten by subsequent calls to *inet_ntoa*( ).

**SEE ALSO**

*endhostent*( ), *endnetent*( ), **<arpa/inet.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

inet_pton, inet_ntop — convert IPv4 and IPv6 addresses between binary and text form.

**SYNOPSIS**

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);

int inet_pton(int af, const char *src, void *dst);
```

**DESCRIPTION**

The *inet_ntop*( ) function converts a numeric address into a text string suitable for presentation. The *af* argument specifies the family of the address. This can be AF_INET or AF_INET6. The *src* argument points to a buffer holding an IPv4 address if the *af* argument is AF_INET, or an IPv6 address if the *af* argument is AF_INET6. The *dst* argument points to a buffer where the function will store the resulting text string; it must not be NULL. The *size* argument specifies the size of this buffer, which must be large enough to hold the text string (INET_ADDRSTRLEN characters for IPv4, INET6_ADDRSTRLEN characters for IPv6).

The *inet_pton*( ) function converts an address in its standard text presentation form into its numeric binary form. The *af* argument specifies the family of the address. The AF_INET and AF_INET6 address families are supported. The *src* argument points to the string being passed in. The *dst* argument points to a buffer into which the function stores the numeric address; this must be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for AF_INET6).

If the *af* argument of *inet_pton*( ) is AF_INET, the *src* string must be in the standard IPv4 dotted-decimal form:

```
ddd.ddd.ddd.ddd
```

where ddd is a one to three digit decimal number between 0 and 255 (see the *inet_addr*( ) definition). The *inet_pton*( ) function does not accept other formats (such as the octal numbers, hexadecimal numbers, and fewer than four numbers that *inet_addr*( ) accepts).

If the *af* argument of *inet_pton*( ) is AF_INET6, the *src* string must be in one of the following standard IPv6 text forms:

1. The preferred form is `x:x:x:x:x:x:x:x`, where the 'x's are the hexadecimal values of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted, but there must be at least one numeral in every field.
2. A string of contiguous zero fields in the preferred form can be shown as "`::`". The "`::`" can only appear once in an address. Unspecified addresses (`0:0:0:0:0:0:0:0`) may be represented simply as "`::`".
3. A third form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is `x:x:x:x:x:x:d.d.d.d`, where the "x"s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the "d"s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).

A more extensive description of the standard representations of IPv6 addresses can be found in referenced document RFC 2373.

**RETURN VALUE**

The *inet_ntop*( ) function returns a pointer to the buffer containing the text string if the conversion succeeds, and NULL otherwise.

The *inet_pton*( ) function returns 1 if the conversion succeeds, with the address pointed to by *dst* in network byte order. It returns 0 if the input is not a valid IPv4 dotted-decimal string or a valid

IPv6 address string, or -1 with *errno* set to [EAFNOSUPPORT] if the *af* argument is unknown.

**ERRORS**

| | |
|---|---|
| [EAFNOSUPPORT] | the *af* argument is invalid |
| [ENOSPC] | the size of the *inet_ntop*( ) result buffer is inadequate. |

**SEE ALSO**

**<arpa/inet.h>**.

**CHANGE HISTORY**

First released in Issue 5.1.

*Chapter 5*

# *IP Address Resolution Headers*

Support for the headers defined in this Chapter is mandatory.

This chapter provides reference manual pages on the headers for the Address Resolution API described in Chapter 4 on page 75.

**NAME**

arpa/inet.h — definitions for internet operations

**SYNOPSIS**

```
#include <arpa/inet.h>
```

**DESCRIPTION**

The <**arpa/inet.h**> header makes available the type **in_port_t** and the type **in_addr_t** as defined in the description of **<netinet/in.h>**.

The <**arpa/inet.h**> header makes available the **in_addr** structure, as defined in the description of **<netinet/in.h>**.

The **<arpa/inet.h>** header makes available the INET_ADDRSTRLEN and INET6_ADDRSTRLEN macros, as defined in the description for **<netinet/in.h>**.

The following may be declared as functions, or defined as macros, or both:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

The **uint32_t** and **uint16_t** types are made available by inclusion of **<inttypes.h>** (see referenced document **XSH**).

The following are declared as functions, and may also be defined as macros:

```
in_addr_t      inet_addr(const char *cp);
in_addr_t      inet_lnaof(struct in_addr in);
struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);
in_addr_t      inet_netof(struct in_addr in);
in_addr_t      inet_network(const char *cp);
char          *inet_ntoa(struct in_addr in);
const char    *inet_ntop(int af, const void *src, char *dst,
                   socklen_t size);
int            inet_pton(int af, const char *src, void *dst);
```

Inclusion of the **<arpa/inet.h>** header may also make visible all symbols from **<netinet/in.h>** and **<inttypes.h>**.

**SEE ALSO**

*htonl*( ), *inet_addr*( ), **<netinet/in.h>**, **<inttypes.h>**.

**CHANGE HISTORY**

First released in Issue 4. Prototypes of *inet_ntop*( ) and *inet_pton*( ) added in Issue 5.2.

**NAME**

netdb.h — definitions for network database operations

**SYNOPSIS**

```
#include <netdb.h>
```

**DESCRIPTION**

The **<netdb.h>** header may make available the type **in_port_t** and the type **in_addr_t** as defined in the description of **<netinet/in.h>**.

The **<netdb.h>** header defines the **hostent** structure that includes at least the following members:

| | |
|---|---|
| `char  *h_name` | Official name of the host. |
| `char **h_aliases` | A pointer to an array of pointers to alternative host names, terminated by a null pointer. |
| `int    h_addrtype` | Address type. |
| `int    h_length` | The length, in bytes, of the address. |
| `char **h_addr_list` | A pointer to an array of pointers to network addresses (in network byte order) for the host, terminated by a null pointer. |

The **<netdb.h>** header defines the **netent** structure that includes at least the following members:

| | |
|---|---|
| `char  *n_name` | Official, fully-qualified (including the domain) name of the host. |
| `char **n_aliases` | A pointer to an array of pointers to alternative network names, terminated by a null pointer. |
| `int    n_addrtype` | The address type of the network. |
| `uint32_t n_net` | The network number, in host byte order. |

The uint32_t type is made available by inclusion of **<inttypes.h>** (see referenced document **XSH**).

The **<netdb.h>** header defines the **protoent** structure that includes at least the following members:

| | |
|---|---|
| `char  *p_name` | Official name of the protocol. |
| `char **p_aliases` | A pointer to an array of pointers to alternative protocol names, terminated by a null pointer. |
| `int    p_proto` | The protocol number. |

The **<netdb.h>** header defines the **servent** structure that includes at least the following members:

| | |
|---|---|
| `char  *s_name` | Official name of the service. |
| `char **s_aliases` | A pointer to an array of pointers to alternative service names, terminated by a null pointer. |
| `int    s_port` | The port number at which the service resides, in network byte order. |
| `char  *s_proto` | The name of the protocol to use when contacting the service. |

The **<netdb.h>** header defines the macro IPPORT_RESERVED with the value of the highest reserved Internet port number.

When the **<netdb.h>** header is included, **h_errno** is available as a modifiable l-value of type **int**. It is unspecified whether **h_errno** is a macro or an identifier declared with external linkage.

The **<netdb.h>** header defines the following macros for use as error values for *gethostbyaddr*( ), *gethostbyname*( ), *getipnodebyaddr*( ), and *getipnodebyname*( )

HOST_NOT_FOUND
NO_DATA
NO_RECOVERY
TRY_AGAIN

The **<netdb.h>** header defines the following macros that evaluate to bitwise-distinct integer constants, for use in the flags argument of *getipnodebyname*( ):

| | |
|---|---|
| AI_V4MAPPED | IPv4-mapped IPv6 addresses are acceptable |
| AI_ALL | return all addresses: IPv6 and IPv4-mapped IPv6 |
| AI_ADDRCONFIG | return addresses depending on what source addresses are configured. |

The **<netdb.h>** header defines macro AI_DEFAULT, which evaluates to the logical OR of AI_V4MAPPED and AI_ADDRCONFIG.

The following are declared as functions, and may also be defined as macros:

```
void             endhostent(void);
void             endnetent(void);
void             endprotoent(void);
void             endservent(void);
void             freehostent(struct hostent *ptr);
struct hostent  *gethostbyaddr(const void *addr, socket_t len, int type);
struct hostent  *gethostbyname(const char *name);
struct hostent  *gethostent(void);
struct hostent  *getipnodebyaddr(const void *addr, socket_t len,
                                 int type, int *error_num);
struct hostent  *getipnodebyname(const char *name, int type, int flags,
                                 int *error_num);
struct netent   *getnetbyaddr(uint32_t net, int type);
struct netent   *getnetbyname(const char *name);
struct netent   *getnetent(void);
struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);
struct protoent *getprotoent(void);
struct servent  *getservbyname(const char *name, const char *proto);
struct servent  *getservbyport(int port, const char *proto);
struct servent  *getservent(void);
void             sethostent(int stayopen);
void             setnetent(int stayopen);
void             setprotoent(int stayopen);
void             setservent(int stayopen);
```

Inclusion of the **<netdb.h>** header may also make visible all symbols from **<netinet/in.h>** and **<inttypes.h>**.

**Address Information Structure**

The **<netdb.h>** header defines the **addrinfo** structure that includes at least the following members:

```
int               ai_flags      Input flags
int               ai_family     Address family of socket
int               ai_socktype   Socket type
int               ai_protocol   Protocol of socket
socklen_t         ai_addrlen    Length of socket address
struct sockaddr  *ai_addr       Socket address of socket
char             *ai_canonname  Canonical name of service location
struct addrinfo  *ai_next       Pointer to next in list
```

The **<netdb.h>** header defines the following macros that evaluate to bitwise-distinct integer constants, for use in the **flags** field of the **addrinfo** structure.

AI_PASSIVE
: socket address is intended for *bind*( )

AI_CANONNAME
: request for canonical name

AI_NUMERICHOST
: return numeric host address as name

The **<netdb.h>** header defines the following macros that evaluate to bitwise-distinct integer constants, for use in the *flags* argument to *getnameinfo*( ).

NI_NOFQDN
: Only the nodename portion of the FQDN is returned for local hosts.

NI_NUMERICHOST
: The numeric form of the node's address is returned instead of its name.

NI_NAMEREQD
: Return an error if the node's name cannot be located in the database.

NI_NUMERICSERV
: The numeric form of the service address is returned instead of its name.

NI_DGRAM
: Indicates that the service is a datagram service (SOCK_DGRAM).

The **<netdb.h>** header defines the following macros for use as error values for *getaddrinfo*( ) and *getnameinfo*( ):

EAI_AGAIN
: The name could not be resolved at this time. Future attempts may succeed.

EAI_BADFLAGS
: The flags had an invalid value

EAI_FAIL
: A non-recoverable error occurred

EAI_FAMILY
: The address family was not recognized or the address length was invalid for the specified family

EAI_MEMORY
: There was a memory allocation failure

EAI_NONAME
: The name does not resolve for the supplied parameters

  NI_NAMEREQD is set and the host's name cannot be located, or both *nodename* and *servname* were null.

EAI_SERVICE
: The service passed was not recognized for the specified socket type

EAI_SOCKTYPE
: The intended socket type was not recognized

EAI_SYSTEM
: A system error occurred. The error code can be found in errno

**SEE ALSO**

*endhostent*( ), *endnetent*( ), *endprotoent*( ), *endservent*( ), *getaddrinfo*( ), *getnameinfo*( ).

**CHANGE HISTORY**

First released in Issue 4.

Address information structure and errors added in Issue 5.2.

**NAME**

netinet/tcp.h — Definitions for the Internet Transmission Control Protocol

**SYNOPSIS**

```
#include <netinet/tcp.h>
```

**DESCRIPTION**

The <netinet/tcp.h> header defines the following macro for use as a socket option at the IPPROTO_TCP level:

| | |
|---|---|
| TCP_NODELAY | Avoid coalescing of small segments |

Note that the macro must be defined in the header, but the implementation need not allow the value of the option to be set via *setsockopt*( ) or retrieved via *getsockopt*( ).

**SEE ALSO**

*getsockopt*( ), *setsockopt*( ), **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

**NAME**

unistd.h — standard symbolic constants and types

**Note:** The **XSH** specification contains the basic definition of this interface. The following additional information pertains to IP Address Resolution.

**DESCRIPTION**

The following is declared as a function and may also be defined as a macro:

```
int gethostname(char *address, socklen_t addresslen);
```

**SEE ALSO**

*gethostname*( ) in the referenced **XSH** specification.

**CHANGE HISTORY**

First released in Issue 4.

Chapter 6

# Use of Sockets for Local UNIX Connections

Support for UNIX-Domain sockets is mandatory.

UNIX domain sockets provide process-to-process communication in a single system.

### Headers

Symbolic constant **AF_UNIX** is defined in **<sys/socket.h>** (see **<sys/socket.h>** on page 68) to identify the UNIX domain address family. Header **<sys/un.h>** (see **<sys/un.h>** on page 106) contains other definitions used in connection with UNIX domain sockets.

The **sockaddr_storage** structure defined in **<sys/socket.h>** is large enough to accommodate a **sockaddr_un** structure (see **<sys/un.h>**) and is aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_un** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_un** structure, the *ss_family* field maps onto the *sun_family* field.

**NAME**

sys/un.h — definitions for UNIX-domain sockets

**SYNOPSIS**

```
#include <sys/un.h>
```

**DESCRIPTION**

The **<sys/un.h>** header defines the **sockaddr_un** structure that includes at least the following members:

```
sa_family_t sun_family     address family
char        sun_path[]     socket pathname
```

The **sockaddr_un** structure is used to store addresses for UNIX domain sockets. Values of this type must be cast to **struct sockaddr** for use with the socket interfaces defined in this document.

The **<sys/un.h>** header defines the type **sa_family_t** as described in **<sys/socket.h>**.

**Note:** The size of sun_path (see **<sys/un.h>**) has intentionally been left undefined. This was done for good reasons. Different implementations have used different sizes. For example, BSD4.3 uses a size of 108. BSD4.4 uses a size of 104. Since most of the implementations today originated from BSD versions, most of the major vendors today use a size that ranges from 92 to 108.

Applications should not assume a particular length for *sun_path* or assume that it can hold _POSIX_PATH_MAX characters (255).

**SEE ALSO**

*bind*( ), *socket*( ), *socketpair*( ).

Chapter 7

# Use of Sockets over Internet Protocols based on IPv4

Support for sockets over Internet Protocols based on IPv4 is mandatory.

This Chapter gives the protocol-specific information that is relevant to the use of sockets in connection with TCP, UDP and ICMP over Version 4 of the Internet Protocol — IPv4.

### Headers

Symbolic constant **AF_INET** is defined in **<sys/socket.h>** (see **<sys/socket.h>** on page 68) to identify the IPv4 Internet address family. Header **<netinet/in.h>** (see **<netinet/in.h>** on page 108) contains other definitions used in connection with IPv4 Internet sockets.

The **sockaddr_storage** structure defined in **<sys/socket.h>** is large enough to accommodate a **sockaddr_in** structure (see **<netinet/in.h>** on page 108) and is aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_in** structure, the **ss_family** field maps onto the **sin_family** field.

**NAME**

netinet/in.h — Internet Protocol family

**SYNOPSIS**

```
#include <netinet/in.h>
```

**DESCRIPTION**

When header file **<netinet.h>** is included, the following types are defined through **typedef**.

**in_port_t** An unsigned integral type of exactly 16 bits.

**in_addr_t** An unsigned integral type of exactly 32 bits.

The **<netinet/in.h>** header defines the **in_addr** structure that includes at least the following member:

```
in_addr_t       s_addr
```

The **<netinet/in.h>** header defines the **sockaddr_in** structure that includes at least the following members:

```
sa_family_t     sin_family
in_port_t       sin_port
struct in_addr  sin_addr
unsigned char   sin_zero[8]
```

The **sockaddr_in** structure is used to store addresses for the Internet protocol family. Values of this type must be cast to **struct sockaddr** for use with the socket interfaces defined in this document.

The **<netinet/in.h>** header defines the type **sa_family_t** as described in **<sys/socket.h>**.

The **<netinet/in.h>** header defines the following macros for use as values of the *level* argument of *getsockopt*( ) and *setsockopt*( ):

| | |
|---|---|
| IPPROTO_IP | IP |
| IPPROTO_ICMP | Control message protocol |
| IPPROTO_TCP | TCP |
| IPPROTO_UDP | User datagram protocol |

The **<netinet/in.h>** header defines the following macros for use as destination addresses for *connect*( ), *sendmsg*( ) and *sendto*( ):

| | |
|---|---|
| INADDR_ANY | IPv4 local host address |
| INADDR_BROADCAST | IPv4 broadcast address |

The **<netinet/in.h>** header defines the following macro to help applications declare buffers of the proper size to store IPv4 addresses in string form:

```
INET_ADDRSTRLEN     16
```

*ntohl*( ), *ntohs*( ), *htonl*( ) and *htons*( ) as defined in the description of **<arpa/inet.h>** are available. Inclusion of the **<netinet/in.h>** header may also make visible all symbols from **<arpa/inet.h>**.

**SEE ALSO**

*getsockopt*( ), *setsockopt*( ). **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 4.

Chapter 8

# Use of Sockets over Internet Protocols based on IPv6

Support for sockets over Internet Protocols based on IPv6 is optional.

This Chapter gives the protocol-specific information that is relevant to the use of sockets in connection with TCP, UDP and ICMP over Version 6 of the Internet Protocol — IPv6. The IPv6 protocol is described in referenced document RFC 2460.

To enable smooth transition from IPv4 to IPv6, the features defined in this Chapter may in certain circumstances also be used in connection with IPv4 - see section Section 8.2 on page 112.

## 8.1 Addressing

IPv6 overcomes the addressing limitations of previous versions, by using 128-bit addresses instead of 32-bit addresses. The IPv6 address architecture is described in referenced document RFC 2373.

There are three kinds of IPv6 address:

Unicast
: Identifies a single interface.

  A unicast address can be global, link-local (designed for use on a single link) or site-local (designed for systems not connected to the Internet). Link-local and site-local addresses need not be globally unique.

Anycast
: Identifies a set of interfaces such that a packet sent to the address can be delivered to any member of the set.

  An anycast address is similar to a unicast address; the nodes to which an anycast address is assigned must be explicitly configured to know that it is an anycast address.

Multicast
: Identifies a set of interfaces such that a packet sent to the address should be delivered to every member of the set.

  An application can send multicast datagrams by simply specifying an IPv6 multicast address in the *address* argument of *sendto*( ). To receive multicast datagrams, an application must join the multicast group (using *setsockopt*( ) with IPV6_JOIN_GROUP) and must bind to the socket the UDP port on which datagrams will be received. Some applications should also bind the multicast group address to the socket, to prevent other datagrams destined to that port from being delivered to the socket.

  A multicast address can be global, node-local, link-local, site-local or organization-local.

The following special IPv6 addresses are defined:

Unspecified
: An address that is not assigned to any interface and is used to indicate the absence of an address.

Loopback
: A unicast address that is not assigned to any interface and can be used by a node to send packets to itself.

Two sets of IPv6 addresses are defined to correspond to IPv4 addresses:

IPv4-compatible addresses
: These are assigned to nodes that support IPv6 and can be used when traffic is "tunneled" through IPv4.

IPv4-mapped addresses
: These are used to represent IPv4 addresses in IPv6 address format. See Section 8.2.

Note that the unspecified address and the loopback address must not be treated as IPv4-compatible addresses.

## 8.2 Compatibility with IPv4

The API provides the ability for IPv6 applications to interoperate with applications using IPv4, by using IPv4-mapped IPv6 addresses. These addresses can be generated automatically by the *getipnodebyname*( ) function when the specified host has only IPv4 addresses (as described in *endhostent*( ) on page 76).

Applications may use AF_INET6 sockets to open TCP connections to IPv4 nodes, or send UDP packets to IPv4 nodes, by simply encoding the destination's IPv4 address as an IPv4-mapped IPv6 address, and passing that address, within a sockaddr_in6 structure, in the *connect*( ), *sendto*( ) or *sendmsg*( ) call. When applications use AF_INET6 sockets to accept TCP connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system returns the peer's address to the application in the *accept*( ), *recvfrom*( ), *recvmsg*( ), or *getpeername*( ) call using a **sockaddr_in6** structure encoded this way. If a node has an IPv4 address, then the implementation may allow applications to communicate using that address via an AF_INET6 socket. In such a case, the address will be represented at the API by the corresponding IPv4-mapped IPv6 address. Also, the implementation may allow an AF_INET6 socket bound to **in6addr_any** to receive inbound connections and packets destined to one of the node's IPv4 addresses.

An application may use AF_INET6 sockets to bind to a node's IPv4 address by specifying the address as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure in the *bind*( ) call. For an AF_INET6 socket bound to a node's IPv4 address, the system returns the address in the *getsockname*( ) call as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure.

## 8.3 Interface Identification

Each local interface is assigned a unique positive integer as numeric index. Indexes start at 1; zero is not used. There may be gaps so that there is no current interface for a particular positive index. Each interface also has a unique system-specific name.

## 8.4 Options

The following options apply at the IPPROTO_IPV6 level:

IPV6_JOIN_GROUP
: When set via *setsockopt*( ), it joins the application to a multicast group on an interface (identified by its index) and addressed by a given multicast address, enabling packets sent to that address to be read via the socket. If the interface index is specified as zero, the system selects the interface (for example, by looking up the address in a routing table and using the resulting interface).

  An attempt to read this option using *getsockopt*( ) results in an errror [EOPNOTSUPP].

  The value of this option is an **ipv6_mreq** structure.

IPV6_LEAVE_GROUP
: When set via *setsockopt*( ), it removes the application from the multicast group on an interface (identified by its index) and addressed by a given multicast address.

  An attempt to read this option using *getsockopt*( ) results in an errror [EOPNOTSUPP].

  The value of this option is an **ipv6_mreq** structure.

IPV6_MULTICAST_HOPS
: The value of this option is the hop limit for outgoing multicast IPv6 packets sent via the socket. Its possible values are the same as those of IPV6_UNICAST_HOPS. If the IPV6_MULTICAST_HOPS option is not set, a value of 1 is assumed. This option can be set via *setsockopt*( ) and read via *getsockopt*( ).

IPV6_MULTICAST_IF
: The index of the interface to be used for outgoing multicast packets. It can be set via *setsockopt*( ) and read via *getsockopt*( ).

IPV6_MULTICAST_LOOP
: This option controls whether outgoing multicast packets should be delivered back to the local application when the sending interface is itself a member of the destination multicast group. If it is set to 1 they are delivered. If it is set to 0 they are not. Other values result in error [EINVAL]. This option can be set via *setsockopt*( ) and read via *getsockopt*( ).

IPV6_UNICAST_HOPS
: The value of this option is the hop limit for outgoing unicast IPv6 packets sent via the socket. If the option is not set, or is set to -1, the system selects a default value. Attempts to set a value less than -1 or greater than 255 result in an [EINVAL] error. This option can be set via *setsockopt*( ) and read via *getsockopt*( ).

Error [EOPNOTSUPP] results if IPV6_JOIN_GROUP or IPV6_LEAVE_GROUP is used with *getsockopt*( ).

## 8.5 Headers

Symbolic constant AF_INET6 is defined in **<sys/socket.h>** to identify the IPv6 Internet address family.

The **sockaddr_storage** structure defined in **<sys/socket.h>** is large enough to accommodate a **sockaddr_in6** structure (see **<netinet/in.h>** on page 108) and is aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in6** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_in6** stricture, the **ss_family** field maps onto the **sin6_family** field.

Headers **<netinet/in.h>**, **<arpa/inet.h>** and **<netdb.h>** contain other definitions used in connection with IPv6 Internet sockets.

**NAME**

netinet/in.h — Internet Protocol family - IP version 6 additions

**NOTE**

Chapter 7 on page 107 (Use of Sockets over Internet Protocols based on IPv4) contains the basic definition of this interface. The following additional information pertains to IPv6.

**DESCRIPTION**

The **<netinet/in.h>** header defines the **in6_addr** structure that contains member **s6_addr[16]**, a 16-element array of *uint8_t*. This array is used to contain a 128-bit IPv6 address, stored in network byte order.

The **<netinet/in.h>** header defines the **sockaddr_in6** structure that includes at least the following members:

| member | type | value |
|---|---|---|
| sin6_family | sa_family_t | AF_INET6 |
| sin6_port | in_port_t | port number |
| sin6_flowinfo | uint32_t | IPv6 traffic class and flow information |
| sin6_addr | struct in6_addr | IPv6 address |
| uint32_t | sin6_scope_id | set of interfaces for a scope |

The **sockaddr_in6** structure should be set to zero by an application prior to using it, since implementations are free to have additional, implementation specific, fields in **sockaddr_in6**.

The *sin6_scope_id* field is a 32-bit integer that identifies a set of interfaces as appropriate for the scope of the address carried in the *sin6_addr* field. For a link scope *sin6_addr*, *sin6_scope_id* would be an interface index. For a site scope *sin6_addr*, *sin6_scope_id* would be a site identifier. The mapping of *sin6_scope_id* to an interface or set of interfaces is left to implementation and future specifications on the subject of site identifiers.

The **<netinet/in.h>** header declares the external variable *in6addr_any* of type **struct in6_addr**. This variable is initialised by the system to contain the wildcard IPv6 address. It also defines symbolic constant IN6ADDR_ANY_INIT which the application can use to initialize a variable of type **struct in6_addr** to the IPv6 wildcard address.

The **<netinet/in.h>** header declares the external variable *in6addr_loopback* of type **struct in6_addr**. This variable is initialised by the system to contain the loopback IPv6 address. It also defines symbolic constant IN6ADDR_LOOPBACK_INIT which the application can use to initialize a variable of type **struct in6_addr** to the IPv6 loopback address.

The **<netinet/in.h>** header defines integer symbolic constant IPPROTO_IPV6 to identify the IPV6 protocol level in *getsockopt*( ) and *setsockopt*( ) calls.

The **<netinet/in.h>** header defines the **ipv6_mreq** structure that includes at least the following members:

| member | type | value |
|---|---|---|
| ipv6mr_multiaddr | struct in6_addr | IPv6 multicast address |
| ipv6mr_interface | unsigned int | interface index |

The **<netinet/in.h>** header defines the following macro to help applications declare buffers of the proper size to store IPv6 addresses in string form:

```
INET6_ADDRSTRLEN    46
```

The **<netinet/in.h>** header defines the following macros, with distinct integral values, for use in the *option_name* argument in *getsockopt*( ) or *setsockopt*( ) calls at protocol level IPPROTO_IPV6:

| | |
|---|---|
| IPV6_JOIN_GROUP | join a multicast group |
| IPV6_LEAVE_GROUP | quit a multicast group |
| IPV6_MULTICAST_HOPS | Multicast hop limit. |
| IPV6_MULTICAST_IF | Interface to use for outgoing multicast packets |
| IPV6_MULTICAST_LOOP | Multicast packets are delivered back to the local application |
| IPV6_UNICAST_HOPS | Unicast hop limit |

The **<netinet/in.h>** header defines the following macros that test for special IPv6 addresses. Each macro is of type int and takes a single argument of type **const struct in6_addr***.

| **Macro** | Description |
|---|---|
| IN6_IS_ADDR_UNSPECIFIED | unspecified address |
| IN6_IS_ADDR_LOOPBACK | loopback address |
| IN6_IS_ADDR_MULTICAST | multicast address |
| IN6_IS_ADDR_LINKLOCAL | unicast link-local address |
| IN6_IS_ADDR_SITELOCAL | unicast site-local address |
| IN6_IS_ADDR_V4MAPPED | IPv4 mapped address |
| IN6_IS_ADDR_V4COMPAT | IPv4 compatible address |
| IN6_IS_ADDR_MC_NODELOCAL | multicast node-local address |
| IN6_IS_ADDR_MC_LINKLOCAL | multicast link-local address |
| IN6_IS_ADDR_MC_SITELOCAL | multicast site-local address |
| IN6_IS_ADDR_MC_ORGLOCAL | multicast organization-local address |
| IN6_IS_ADDR_MC_GLOBAL | multicast global address |

Note that IN6_IS_ADDR_LINKLOCAL and IN6_IS_ADDR_SITELOCAL return true only for the two local-use IPv6 unicast addresses. They do not return true for multicast addresses of either link-local or site-local scope.

**SEE ALSO**

*getsockopt*( ), *setsockopt*( ). **<sys/socket.h>**.

**CHANGE HISTORY**

First released in Issue 5.1.

*Technical Standard*

# Networking Services (XNS) Issue 5.2

## Part 3: XTI

*The Open Group*

Chapter 9

# General Introduction to the XTI

Support for XTI as defined in this Part 3 of the XNS Technical Standard is optional. The XTI interface is obsolete. Writers of new applications using the Internet protocol suite are recommended to use sockets rather than XTI. Where protocols for which there is no sockets support are in use, XTI is still recommended in preference to proprietary APIs.

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

- connection establishment
- state change support
- event handling
- data transfer
- option manipulation.

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

The main Chapters in this **Part 2: XTI** specification describe interfaces, parameters and semantics constant across all transport providers. Several Appendices provide information that is not an integral part of the main body since it is either descriptive or applies only to some transport providers.

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

- Those corresponding to traditional transport providers, such as:
  - ISO Transport (connection-mode or connectionless-mode)
  - TCP
  - UDP
  - NetBIOS.
- Those corresponding to commonly used subsets of higher-layer protocols that provide transport-like services, such as:
  - minimal functionality OSI (mOSI), that is, OSI ACSE/Presentation with the kernel and duplex functional units
  - SNA LU6.2 subset.
- Mixed-protocol providers that provide the appearance of one protocol over a different protocol such as:
  - ISO transport appearance (connection-mode) over TCP.

    The ISO appendix (Appendix A) also describes a transport provider that uses RFC 1006 to compensate for the differences between ISO transport and TCP so that a TCP provider

can present an ISO transport appearance.

While XTI gives transport users considerable independence from the underlying transport provider, the differences between providers are not entirely hidden. Appendix B on page 279 includes guidelines for writing transport-provider-independent software, which can be done primarily by using only functions supported by all providers, avoiding option management, and using a provider-independent means of acquiring addresses.

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

Some of the Appendices to the XTI specification are included as vehicles for communicating information needed by implementors, or guidelines to the use of the specification in question. The Guidelines for the use of XTI (see Appendix B on page 279), Minimum OSI Functionality (see Appendix F on page 331), (Appendix H), SNA transport provider (see Appendix G on page 353), SPX/IPX transport provider (see Appendix H on page 387), and Comparison of XTI to TLI (see all belong to this category.

Some other Appendices, however, have evolved into a prescriptive specification, as in the case of the ISO transport provider (see Appendix A on page 265), and the NetBIOS transport provider (see Appendix C on page 305).

Since not every XTI implementor would find it relevant to implement the functionality of all of these Appendices, they have been kept separate from the definitions for XTI. Thus they are readily identifable as brandable XTI options. Support for these transport providers is declared in Branding documentation through the XTI Conformance Statement Questionnaire.

Topics beyond the scope of the XTI specification include:

- Address parameters

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

- Event management

  In order for applications to use XTI in a fully asynchronous manner, it will be necessary for the application to include facilities of an Event Management interface. Such an event management facility may allow the application to be notified of a number of events over a range of active transport connections. For example, one event may denote a connection is flow-controlled. While Section B.5 on page 284 provides some guidelines for using event management in XTI applications, a complete specification defining an event management interface is beyond the scope of this document.

Chapter 10

# Explanatory Notes for XTI

## 10.1 Transport Endpoints

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

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

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

## 10.2 Transport Providers

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

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

## 10.3 Association of a UNIX Process to an Endpoint

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

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

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

## 10.4 Use of the Same Protocol Address

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

## 10.5 Modes of Service

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

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

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

## 10.6 Error Handling

Two levels of error are defined for the transport interface. The first is the library error level. Each library function has one or more error returns. Failures are indicated by a return value of –1. When header file **<xti.h>** is included, symbol *t_errno* is defined as a modifiable *lvalue* of type **int, t_errno,**[5] and can be used to access the specific error number when such a failure occurs. Applications should not include *t_errno* in the left operand of assignment statements. This value is set when errors occur but is not cleared on successful library calls, so it should be tested only after an error has been indicated. A diagnostic function, *t_error*( ), prints out information on the current transport error. The state of the transport provider may change if a transport error occurs.

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

For example, a system error can be generated by the transport provider when a protocol error has occurred. If the error is severe, it may cause the file descriptor and transport endpoint to be

5. This may be implemented as a macro. In addition the name **_t_errno** is an XTI library-reserved-name for use within such a macro. A typical definition of *t_errno* for a multithreaded implementation is:

```
extern int *_t_errno(void);
#define t_errno (*(_t_errno()))
```

unusable. To continue in this case, all users of the *fd* must close it. Then the transport endpoint may be re-opened and initialised.

## 10.7 Synchronous and Asynchronous Execution Modes

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

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

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

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

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

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

These are described in Section 10.9 on page 126.

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

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

**Events and t_look( )**

All events that occur at a transport endpoint are stored by XTI. These events are retrievable one at a time via the *t_look*( ) function. If multiple events occur, it is implementation-dependent in what order *t_look*( ) will return the events. An event is outstanding on a transport endpoint until it is consumed. Every event has a corresponding consuming function which handles the event and consumes it. In addition, the abortive T_DISCONNECT consumes other pending events. Both T_DATA and T_EXDATA events are consumed when the corresponding consuming function has read all the corresponding data associated with that event. The intention of this is that T_DATA should always indicate that there is data to receive. Two events, T_GODATA and T_GOEXDATA, are also cleared as they are returned by *t_look*( ). Table 10-1 summarises this.

| **Event** | **Cleared on *t_look*( ) ?** | **Consuming XTI functions** |
|---|---|---|
| T_LISTEN | No | *t_listen*( ) |
| T_CONNECT | No | *t_{rcv}connect*( )[7] |
| T_DATA | No | *t_rcv{v}{udata}*( ) |
| T_EXDATA | No | *t_rcv{v}*( ) |
| T_DISCONNECT | No | *t_rcvdis*( ) |
| T_UDERR | No | *t_rcvuderr*( ) |
| T_ORDREL | No | *t_rcvrel{data}*( ) |
| T_ORDRELDATA | No | *t_rcvreldata*( ) |
| T_GODATA | Yes | *t_snd{v}{udata}*( ) |
| T_GOEXDATA | Yes | *t_snd{v}*( ) |

**Table 10-1** Events and *t_look*( )

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

## 10.8 Effect of Signals

In both the synchronous and the asynchronous execution modes, XTI calls may be affected by signals. Unless specified otherwise in the description of each function, the functions behave as described below.

If a synchronous XTI call is blocking under circumstances where an asynchronous call would have returned because no event was available, then the call returns –1 with *t_errno* set to [TSYSERR] and *errno* set to [EINTR]. The state of the endpoint is unchanged.

In addition an [EINTR] error may be returned by all XTI calls (except *t_error*( ) and *t_strerror*( )) under implementation defined conditions. In these cases the state of the endpoint will not have been changed, and no data will have been sent or received. Any buffers provided by the user for return values may have been overwritten.

A ''well written'' application will itself mask out signals except during specific code sequences (typically only its idle point) to avoid having to handle an [EINTR] return from all system calls.

Application writers should be aware that XTI calls may be implemented in a library as multiple system calls. In order to maintain the endpoint and associated library data areas in a consistent state, some of these system calls may be repeated when interrupted by a signal.

Applications should not call XTI functions from within a signal handler or using the *longjmp*( ) or *siglongjmp*( ) interfaces (see reference **XSH**) to exit a signal handler, as either may leave XTI data areas in an inconsistent state.

Applications may be able to cause the XTI library itself to generate signals that interrupt its internal actions (for example, by issuing *ioctl( fd, I_SETSIG, S_INPUT )* on a UNIX system); this may cause the user's signal handler to be scheduled, but will not stop the XTI call from completing.

## 10.9 Event Management

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

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

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

1. The file descriptor is bound to a valid address.
2. No transport connection is established at this time.

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

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

T_EXDATA — Expedited data was received by the transport provider.

| | |
|---|---|
| T_DISCONNECT | In connection mode only; a disconnection request was received by the transport provider. It may be reported on both data transfer functions and connection establishment functions and on the *t_snddis*( ) function. |
| T_ORDREL | An orderly release request was received by a transport provider (connection mode with orderly release only). |
| T_UDERR | In connectionless-mode only; an error was found in a previously sent datagram. It may be notified on the *t_rcvudata*( ), *rcvvudata*( ), or *t_unbind*( ) function calls. |
| T_GODATA | Flow control restrictions on normal data flow that led to a [TFLOW] error have been lifted. Normal data may be sent again. |
| T_GOEXDATA | Flow control restrictions on expedited data flow that led to a [TFLOW] error have been lifted. Expedited data may be sent again. |

Chapter 11

# XTI Overview

## 11.1 Overview of Connection-oriented Mode

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

- Initialisation / De-initialisation
- Connection Establishment
- Data Transfer
- Connection Release.

A state machine is described in Section B.1 on page 279, and the figure in Section B.2 on page 280, which defines the legal sequence in which functions from each phase may be issued.

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

1. supply a *transport provider identifier* for the appropriate type of transport provider (using *t_open*( ) ); this establishes a transport endpoint through which the user may communicate with the provider
2. associate (bind) an address with this endpoint (using *t_bind*( ) )
3. use the appropriate connection functions (using *t_connect*( ), or *t_listen*( ) and *t_accept*( ) ) to establish a transport connection; the set of functions depends on whether the user is an initiator or responder
4. once the connection is established, normal, and if authorised, expedited data can be exchanged; of course, expedited data may be exchanged only if:
   - the provider supports it
   - its use is not precluded by the selection of protocol characteristics; for example, the use of ISO Transport Class 0
   - both the peer tranport providers support the feature, have negotiated it if necessary, and the peer applications have agreed to its use.

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

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

When reusing an endpoint, the user should be aware that the local address may be different from that originally bound (for example, one of the systems' local addresses may be explicitly associated with the endpoint, or the address may have been changed during *t_accept*( )). The value of certain options will also reflect the previous connect and may need to be reset (see the

definition for *t_optmgmt*( )) to the providers' default values.

### 11.1.1 Initialisation/De-initialisation Phase

The functions that support initialisation/de-initialisation tasks are described below. All such functions provide local management functions.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

*t_sysconf*( ) This function is used to obtain the values of configurable and implementation-dependent XTI variables.

### 11.1.2 Overview of Connection Establishment

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

In connection mode:

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

  If the same endpoint is used, the listening endpoint cannot receive or enqueue incoming connect indications for the duration of the connection.

  If a different endpoint is used, the listening endpoint can continue to receive and enqueue incoming connect indications.
- If the user issues a *t_connect*( ) on a listening endpoint, again, that endpoint cannot receive or enqueue incoming connect indications for the duration of the connection.
- A connection attempt failure will result in a value –1 returned from either the *t_connect*( ) or *t_rcvconnect*( ) call, with *t_errno* set to [TLOOK] indicating that a [T_DISCONNECT] event has arrived. In this case, the reason for the failure may be identified by issuing a *t_rcvdis*( ) call.

The functions that support these operations of connection establishment are:

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

response arrives.

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

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

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

### 11.1.3 Overview of Data Transfer

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

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

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

*t_sndv*( ) This function enables transport users to send either normal or expedited data from non-contiguous buffers over a transport connection.

*t_rcvv*( ) This function enables transport users to receive either normal or expedited data into non-contiguous buffers on a transport connection.

Throughout the rest of this section, all references of calls to *t_rcv*( ) include calls to *t_rcvv*( ), and calls to *t_snd*( ) include calls to *t_sndv*( ).

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

#### Receiving Data

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

- Asynchronous Mode

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

- Synchronous Mode

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

  — Data (normal or expedited) is received.

  — A disconnection indication is received.

— A signal has arrived.

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

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

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

**Sending Data**

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

- Asynchronous Mode

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

- Synchronous Mode

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

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

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

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

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

Note that XTI supports two modes of sending data, record-oriented and stream-oriented. In the record-oriented mode, the concept of TSDU is supported, that is, message boundaries are preserved. In stream-oriented mode, message boundaries are not preserved and the concept of a TSDU is not supported. A transport user can determine the mode by using the *t_getinfo*( ) function, and examining the *tsdu* field. If *tsdu* is greater than zero, this indicates that record-oriented mode is supported and the return value indicates the maximum TSDU size. If *tsdu* is zero, this indicates that stream-oriented transfer is supported. For more details see *t_getinfo*( ) on page 182.

### 11.1.4 Overview of Connection Release

Some communication providers (for example, the ISO connection-mode transport) support only the abortive release. However, some communication providers (for example, mOSI, NetBIOS, SNA, TCP) also support an orderly release. XTI includes functions to provide access to transports that support or require the orderly release features.

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

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

Some transport providers support an orderly release capability (for example, mOSI, NetBIOS, SNA, TCP). If supported by the communications provider, orderly release may be invoked from the data transfer phase to enable two users to gracefully release a connection. The procedure for orderly release prevents the loss of data that may occur during an abortive release.

When supported by the underlying protocol, some communications providers optionally allow applications to send or retrieve user data with an orderly release, through the use of the *t_sndreldata*( ) or *t_rcvreldata*( ) functions instead of the *t_sndrel*( ) or *t_rcvrel*( ) functions.

The functions that support connection release are:

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

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

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

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

*t_sndreldata*( ) This function can be used instead of *t_sndrel*( ) to send user data with an orderly release.

*t_rcvreldata*( ) This function can be used instead of *t_rcvrel*( ) to retrieve orderly release user data.

## 11.2 Overview of Connectionless Mode

The connectionless-mode transport service consists of two phases of communication: initialisation/de-initialisation and data transfer. A brief description of each phase and its associated functions is presented below. A state machine is described in Section B.1 on page 279, and the figure in Section B.3 on page 282, that defines the legal sequence in which functions from each phase may be issued.

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

1. supply a transport endpoint for the appropriate type of provider (using *t_open*( ) ); this establishes a transport endpoint through which the user may communicate with the provider
2. associate (bind) an address with this transport endpoint (using *t_bind*( ) )

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

### 11.2.1 Initialisation/De-initialisation Phase

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

### 11.2.2 Overview of Data Transfer

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

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

*t_sndvudata*( ) This function enables transport users to send a self-contained data unit to the user from one or more non-contiguous buffers at the specified protocol address.

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

*t_rcvvudata*( ) This function enables transport users to receive data units from other users into one or more non-contiguous buffers.

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

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

Throughout the rest of this section, all references of calls to *rcvudata*( ) include calls to *rcvvudata*( ), and calls to *t_sndudata*( ) include calls to *t_sndvudata*( ).

**Receiving Data**

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

- Asynchronous Mode

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

- Synchronous Mode

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

  — A datagram is received.

  — An error is detected by the transport provider.

  — A signal has arrived.

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

**Sending Data**

- Synchronous Mode

  In order to maintain some flow control, the *t_sndudata*( ) function will block until the datagram has been accepted by the provider. The call returns immediately after the datagram has been sent. A process which sends data in synchronous mode may be blocked for some time.

- Asynchronous Mode

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

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

## 11.3 XTI Features

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

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

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

*t_accept*( )
*t_connect*( )
*t_listen*( )
*t_rcv*( )
*t_rcvv*( )
*t_rcvconnect*( )
*t_rcvdis*( )
*t_snd*( )
*t_sndv*( )
*t_snddis*( )

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

*t_rcvudata*( )
*t_rcvvudata*( )
*t_rcvuderr*( )
*t_sndudata*( )
*t_sndvudata*( )

**Mandatory mechanisms:**

- synchronous mode
- asynchronous mode.

**Utility functions:**

*t_alloc*( )
*t_free*( )
*t_error*( )
*t_getprotaddr*( )
*t_getinfo*( )
*t_getstate*( )
*t_optmgmt*( )
*t_strerror*( )
*t_sysconf*( )

The orderly release mechanism (using *t_sndrel*( ), *t_sndreldata*( ), *t_rcvrel*( ) and *t_rcvreldata*( )) is supported only for T_COTS_ORD type providers. Use with other providers will cause the [TNOTSUPPORT] error to be returned.

**Optional mechanisms:**

- the ability to manage (enqueue) more than one incoming connection indication at any one time
- the address of the caller passed with *t_accept*( ) may optionally be checked by an XTI implementation

### 11.3.1 XTI Functions versus Protocols

Table 11-1 presents all the functions defined in XTI. The character ‘‘x’’ indicates that the mapping of that function is possible onto a connection-mode or connectionless-mode Transport Service. The table indicates the type of utility functions as well.

| Functions | Necessary for Protocol | | Utility Functions | |
|---|---|---|---|---|
| | **Connection Mode** | **Connectionless Mode** | **General** | **Memory** |
| *t_accept*( ) | x | | | |
| *t_alloc*( ) | | | | x |
| *t_bind*( ) | x | x | | |
| *t_close*( ) | x | x | | |
| *t_connect*( ) | x | | | |
| *t_error*( ) | | | x | |
| *t_free*( ) | | | | x |
| *t_getprotaddr*( ) | | | x | |
| *t_getinfo*( ) | | | x | |
| *t_getstate*( ) | | | x | |
| *t_listen*( ) | x | | | |
| *t_look*( ) | x | x | | |
| *t_open*( ) | x | x | | |
| *t_optmgmt*( ) | | | x | |
| *t_rcv*( ) | x | | | |
| *t_rcvv*( ) | x | | | |
| *t_rcvconnect*( ) | x | | | |
| *t_rcvdis*( ) | x | | | |
| *t_rcvrel*( ) | x | | | |
| *t_rcvreldata*( ) | x | | | |
| *t_rcvudata*( ) | | x | | |
| *t_rcvvudata*( ) | | x | | |
| *t_rcvuderr*( ) | | x | | |
| *t_snd*( ) | x | | | |
| *t_sndv*( ) | x | | | |
| *t_snddis*( ) | x | | | |
| *t_sndrel*( ) | x | | | |
| *t_sndreldata*( ) | x | | | |
| *t_sndudata*( ) | | x | | |
| *t_sndvudata*( ) | | x | | |
| *t_strerror*( ) | | | x | |
| *t_sync*( ) | | | x | |
| *t_sysconf*( ) | | | x | |
| *t_unbind*( ) | x | x | | |

**Table 11-1** Classification of the XTI Functions

Chapter 12

# States and Events in XTI

Table 12-1 through Table 12-7 are included to describe the possible states of the transport provider as seen by the transport user, to describe the incoming and outgoing events that may occur on any connection, and to identify the allowable sequence of function calls. Given a current state and event, the transition to the next state is shown as well as any actions that must be taken by the transport user.

The allowable sequence of functions is described in Table 12-5, Table 12-6 and Table 12-7. The support functions, *t_getprotaddr*( ), *t_getstate*( ), *t_getinfo*( ), *t_alloc*( ), *t_free*( ), *t_look*( ) and *t_sync*( ), are excluded from the state tables because they do not affect the state of the interface. Each of these functions may be issued from any state except the uninitialised state. Similarly, the *t_error*( ), *t_strerror*( ) and *t_sysconf*( ) functions have been excluded from the state table because they do not affect the state of the interface.

## 12.1 Transport Interfaces States

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

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

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

Table 12-1 describes all possible states of the transport provider as seen by the transport user. The service type may be connection-mode, connection-mode with orderly release or connectionless-mode.

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

**Table 12-1** Transport Interface States

## 12.2 Outgoing Events

The following outgoing events correspond to the successful return or error return of the specified user-level transport functions causing XTI to change state, where these functions send a request or response to the transport provider. In Table 12-2, some events (for example, accept1, accept2 and accept3) are distinguished by the context in which they occur. The context is based on the values of the following:

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

*fd* File descriptor of the current transport endpoint.

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

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

**Table 12-2** Transport Interface Outgoing Events

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

## 12.3 Incoming Events

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

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

In Table 12-3, the *rcvdis* events are distinguished by the context in which they occur. The context is based on the value of *ocnt*, which is the count of outstanding connection indications on the current transport endpoint.

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

**Table 12-3** Transport Interface Incoming Events

## 12.4 Transport User Actions

Some state transitions are accompanied by a list of actions the transport user must take. These actions are represented by the notation [*n*], where *n* is the number of the specific action as described in Table 12-4.

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

**Table 12-4** Transport Interface User Actions

## 12.5 State Tables

Table 12-5, Table 12-6 and Table 12-7 describe the possible next states, given the current state and event. The state is that of the transport provider as seen by the transport user.

The contents of each box represent the next state given the current state (column) and the current incoming or outgoing event (row). An empty box represents a state/event combination that is invalid. Along with the next state, each box may include an action list (as specified in Table 12-4 on page 145). The transport user must take the specific actions in the order specified in the state table.

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

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

**Table 12-5** Initialisation/De-initialisation States

| **state event** | **T_IDLE** |
|---|---|
| sndudata | T_IDLE |
| rcvudata | T_IDLE |
| rcvuderr | T_IDLE |

**Table 12-6** Data Transfer States: Connectionless-mode

| state<br>event | T_UNBND | T_IDLE | T_OUTCON | T_INCON | T_DATAXFER | T_OUTREL | T_INREL |
|---|---|---|---|---|---|---|---|
| *connect1* | | T_DATAXFER | | | | | |
| *connect2* | | T_OUTCON | | | | | |
| *rcvconnect* | | | T_DATAXFER | | | | |
| *listen* | | T_INCON[2] | | T_INCON[2] | | | |
| *accept1* | | | | T_DATAXFER[3] | | | |
| *accept2* | | | | T_IDLE[3][4] | | | |
| *accept3* | | | | T_INCON[3][4] | | | |
| *snd* | | | | | T_DATAXFER | | T_INREL |
| *rcv* | | | | | T_DATAXFER | T_OUTREL | |
| *snddis1* | | | T_IDLE | T_IDLE[3] | T_IDLE | T_IDLE | T_IDLE |
| *snddis2* | | | | T_INCON[3] | | | |
| *rcvdis1* | | | T_IDLE | | T_IDLE | T_IDLE | T_IDLE |
| *rcvdis2* | | | | T_IDLE[3] | | | |
| *rcvdis3* | | | | T_INCON[3] | | | |
| *sndrel* | | | | | T_OUTREL | | T_IDLE |
| *rcvrel* | | | | | T_INREL | T_IDLE | |
| *pass_conn* | T_DATAXFER | T_DATAXFER | | | | | |
| *optmgmt* | T_UNBND | T_IDLE | T_OUTCON | T_INCON | T_DATAXFER | T_OUTREL | T_INREL |
| *closed* | | T_UNINIT | T_UNINIT | T_UNINIT | T_UNINIT | T_UNINIT | T_UNINIT |

**Table 12-7** Connection / Release / Data Transfer States: Connection-mode

## 12.6 Events and TLOOK Error Indication

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

| | |
|---|---|
| *t_accept*( ) | T_DISCONNECT, T_LISTEN |
| *t_connect*( ) | T_DISCONNECT, T_LISTEN[8] |
| *t_listen*( ) | T_DISCONNECT[9] |
| *t_rcv*( ) | T_DISCONNECT, T_ORDREL[10] |
| *t_rcvconnect*( ) | T_DISCONNECT |
| *t_rcvrel*( ) | T_DISCONNECT |
| *t_rcvreldata*( ) | T_DISCONNECT |
| *t_rcvudata*( ) | T_UDERR |
| *t_rcvv*( ) | T_DISCONNECT, T_ORDREL |
| *t_rcvvudata*( ) | T_UDERR |
| *t_snd*( ) | T_DISCONNECT, T_ORDREL |
| *t_sndudata*( ) | T_UDERR |
| *t_sndv*( ) | T_DISCONNECT, T_ORDREL |
| *t_sndvudata*( ) | T_UDERR |
| *t_unbind*( ) | T_LISTEN, T_DATA[11], T_UDERR |
| *t_sndrel*( ) | T_DISCONNECT |
| *t_sndreldata*( ) | T_DISCONNECT |
| *t_snddis*( ) | T_DISCONNECT |

Once a [TLOOK] error has been received on a transport endpoint via an XTI function, subsequent calls to that and other XTI functions, to which the same [TLOOK] error applies, will continue to return [TLOOK] until the event is consumed. An event causing the [TLOOK] error can be determined by calling *t_look*( ) and then can be consumed by calling the corresponding consuming XTI function as defined in Table 10-1.

---

8. This occurs only when a *t_connect* is done on an endpoint which has been bound with a *qlen* > 0 and for which a connection indication is pending.
9. This event indicates a disconnection on an outstanding connection indication.
10. This occurs only when all pending data has been read.
11. T_DATA may only occur for the connectionless mode.

Chapter 13

# The Use of Options in XTI

## 13.1 Generalities

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

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

Options values have meaning to, and are defined by, the protocol level in which they apply. In some cases, option values can be negotiated by a transport user. This includes the simple case where the transport user can simply enforce its use. Often, the transport provider or even the remote transport user may have the right to negotiate a value of lesser quality than a proposed one, that is, a delay may become longer, or a throughput may become lower.

It is useful to differentiate between options with end-to-end significance (sometimes referred to as *association-related*[12] and those that are not. Association-related options are intimately related to the particular transport connection or datagram transmission. If the calling user specifies such an option, some ancillary information is transferred across the network in most cases. The interpretation and further processing of this information is protocol-dependent. For instance, in an ISO connection-mode communication, the calling user may specify quality-of-service parameters on connection establishment. These are first processed and possibly lowered by the local transport provider, then sent to the remote transport provider that may degrade them again, and finally conveyed to the called user that makes the final selection and transmits the selected values back to the caller.

Options without end-to-end significance do not contain information destined for the remote transport user. Some have purely local relevance, for example, an option that enables debugging. Others influence the transmission, for instance the option that sets the T_IP *time-to-live* field, or T_TCP_NODELAY (see Chapter 16 on page 251). Local options are negotiated solely between the transport user and the local transport provider. The distinction between these two categories of options is visible in XTI through the following relationship: on output, the functions *t_listen*( ) and *t_rcvudata*( ) return options with end-to-end significance only. The functions *t_rcvconnect*( ) and *t_rcvuderr*( ) may return options of both categories. On input, options of both categories may be specified with *t_accept*( ) and *t_sndudata*( ). The functions *t_connect*( ) and *t_optmgmt*( ) can process and return both categories of options.

12. The term ‘‘association’’ is used to denote a pair of communicating transport users, that is, the communication has end-to-end significance.

Options can be further distinguished between those which maintain a persistent value and those which are requests to perform actions to which the option values are effectively parameters. For each action that it supports and that maintains a persistent value, the transport provider has a default value. These defaults are sufficient for the majority of communication relations. Hence, a transport user should only request options actually needed to perform the task, and leave all others at their default value.

This chapter describes the general framework for the use of options. This framework is obligatory for all transport providers. The specific options that are legal for use with a specific transport provider are described in the provider-specific appendices (see Appendix A on page 265 and Chapter 16 on page 251). General XTI options are described in *t_optmgmt*( ) on page 195.

## 13.2 The Format of Options

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

A transport provider embodies a stack of protocols. The *level* field of **struct t_opthdr** identifies the XTI level or a protocol of the transport provider such as T_TCP or ISO 8073:1986. The *name* field identifies the option within the level, and *len* contains its total length, that is, the length of the option header **t_opthdr** plus the length of the option value. The *status* field is used by the XTI level or the transport provider to indicate success or failure of a negotiation (see Section 13.3.5 on page 154 and *t_optmgmt*( ) on page 195).

Several options can be concatenated. The option user has, however to ensure that each options header and value part starts at a boundary appropriate for the architecture-specific alignment rules. The macros T_OPT_FIRSTHDR(nbp), T_OPT_NEXTHDR(nbp, tohp), T_OPT_DATA(tohp) are provided for that purpose. These macros are defines in the man-page for *t_optmgmt*( )( see *t_optmgmt*( ) on page 195).

T_OPT_FIRSTHDR is useful for finding an appropriately aligned start of the option buffer. T_OPT_NEXTHDR is useful for moving to the start of the next appropriately aligned option in the option buffer. Note that OPT_NEXTHDR is also available for backward compatibility requirements. T_OPT_DATA is useful for finding the start of the data part in the option buffer where the contents of its values start on an appropriately aligned boundary.

The length of the option buffer is given by *opt.len*. The alignment characters are included in the length.

## 13.3 The Elements of Negotiation

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

### 13.3.1 Multiple Options and Options Levels

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

### 13.3.2 Illegal Options

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

- The length specified in *t_opthdr.len* exceeds the remaining size of the option buffer (counted from the beginning of the option).
- The option value is illegal. The legal values are defined for each option. (See *t_optmgmt*( ) on page 195 and the protocol specific appendices where options are described for specific protocols).

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

- If an illegal option is passed to *t_optmgmt*( ) then the function fails with *t_errno* set to [TBADOPT].
- If an illegal option is passed to *t_accept*( ) or *t_connect*( ) then either the function fails with *t_errno* set to [TBADOPT] or the connection establishment fails at a later stage, depending on when the implementation detects the illegal option.
- If an illegal option is passed to *t_sndudata*( ) then either the function fails with *t_errno* set to {TBADOPT] or it successfully returns but a T_UDERR event occurs to indicate that the datagram was not sent.

If the transport user passes multiple options in one call and one of them is illegal, the call fails as described above. It is, however, possible that some or even all of the submitted legal options were successfully negotiated. The transport user can check the current status by a call to *t_optmgmt*( ) with the T_CURRENT flag set (see *t_optmgmt*( ) on page 195).

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

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

### 13.3.3 Initiating an Option Negotiation

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

The negotiation rules for these functions depend on whether an option request is an absolute requirement or not. This is explicitly defined for each option (see *t_optmgmt*( ) on page 195 and the protocol specific appendices where options are described for specific protocols). In case of an ISO transport provider, for example, the option that requests use of expedited data is not an absolute requirement. On the other hand, the option that requests protection could be an absolute requirement.

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

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

- The negotiated value is the same as the proposed one. When the result of the negotiation is retrieved, the *status* field in **t_opthdr** is set to T_SUCCESS.
- The negotiation is rejected if the option is supported but the proposed value cannot be negotiated. This leads to the following behaviour:

  — *t_optmgmt*( ) successfully returns, but the returned option has its *status* field set to T_FAILURE.

  — Any attempt to establish a connection aborts; a T_DISCONNECT event occurs, and a synchronous call to *t_connect*( ) fails with [TLOOK].

  — *t_sndudata*( ) fails with [TLOOK] or successfully returns, but a T_UDERR event occurs to indicate that the datagram was not sent.

  If multiple options are submitted in one call and one of them is rejected, XTI behaves as just described. Although the connection establishment or the datagram transmission fails, options successfully negotiated before some option was rejected retain their negotiated values. There is no roll-back mechanism (see Section 13.4 on page 156).

  The function *t_optmgmt*( ) attempts to negotiate each option. The *status* fields of the returned options indicate success (T_SUCCESS) or failure (T_FAILURE).
- If the local transport provider does not support the option at all, *t_optmgmt*( ) reports T_NOTSUPPORT in the *status* field. The functions *t_connect*( ) and *t_sndudata*( ) ignore this option.

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

- The negotiated value is of equal or lesser quality than the proposed one (for example, a delay may become longer).

  When the result of the negotiation is retrieved, the *status* field in **t_opthdr** is set to T_SUCCESS if the negotiated value equals the proposed one, or set to T_PARTSUCCESS otherwise.
- If the local transport provider does not support the option at all, *t_optmgmt*( ) reports T_NOTSUPPORT in the *status* field. The functions *t_connect*( ) and *t_sndudata*( ) ignore this option.

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

for the communication is left to the transport user.

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

Not all options are independent of one another. A requested option value might conflict with the value of another option that was specified in the same call or is currently effective (see Section 13.4 on page 156). These conflicts may not be detected at once, but later they might lead to unpredictable results. If detected at negotiation time, these conflicts are resolved within the rules stated above. The outcomes may thus be quite different and depend on whether absolute or non-absolute requests are involved in the conflict.

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

When called, the functions *t_connect*( ) and *t_sndudata*( ) initiate a negotiation of *all* options with end-to-end significance according to the rules of this section. Options not explicitly specified in the function calls themselves are taken from an internal option buffer that contains the values of a previous negotiation (see Section 13.4 on page 156).

### 13.3.4 Responding to a Negotiation Proposal

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

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

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

If multiple options are submitted with *t_accept*( ) and one of them is rejected, the connection fails as described above. Options that could be successfully negotiated before the erroneous option was processed retain their negotiated value. There is no roll-back mechanism (see Section 13.4 on page 156).

The response options can be specified with the *t_accept*( ) call. Alternatively, they can be specified by calling *t_optmgmt*( ) and passing it the file descriptor that will subsequently be passed as *resfd* to *t_accept*( ) to identify the responding endpoint (see Section 13.4 on page 156. In case of conflict between option settings made by calls to *t_optmgmt*( ) and *t_accept*( ) at different times, the latest settings when *t_accept*( ) is called shall prevail. Note that the response to a negotiation proposal is activated when *t_accept*( ) is called. A *t_optmgmt*( ) call with erroneous option values as described above shall succeed; the connection aborts at the time *t_accept*( ) is called.

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

The function *t_accept*( ) does not check for multiple specification of an option (see Section 13.3.3 on page 152). Unsupported options are ignored.

### 13.3.5 Retrieving Information about Options

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

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

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

Which options are returned depend on the function call involved:

*t_connect*( ) (synchronous mode) and *t_rcvconnect*( )
: The functions return the values of all options with end-to-end significance that were received with the connection response and the negotiated values of those options without end-to-end significance that had been specified on input. However, options specified on input in the *t_connect*( ) call that are not supported or refer to an unknown option level are discarded and not returned on output.

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

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

  The number of received options may be variable for subsequent connection indications, since many options with end-to-end significance are only transmitted on explicit demand by the calling user (for example, T_IP options or ISO 8072:1986 throughput). It is even possible that no options at all are returned.

  The *status* field is irrelevant.

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

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

The *status* field is irrelevant.

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

The *status* field is irrelevant.

*t_optmgmt*( ) This call can process and return both categories of options. It acts on options related to the specified transport endpoint, not on options related to a connection indication or an incoming datagram. A detailed description is given in *t_optmgmt*( ) on page 195.

### 13.3.6 Privileged and Read-only Options

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

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

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

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

- *t_optmgmt*( ) successfully returns, but the returned option has its *status* field set to T_NOTSUPPORT if a privileged option was requested illegally, and to T_READONLY if modification of a read-only option was requested.
- If negotiation of a read-only option is requested, *t_accept*( ) or *t_connect*( ) either fail with [TACCES], or the connection establishment aborts and a T_DISCONNECT event occurs. If the connection aborts, a synchronous call to *t_connect*( ) fails with [TLOOK]. It depends on timing and implementation conditions whether a *t_accept*( ) call still succeeds or fails with [TLOOK].

  If a privileged option is illegally requested, the option is quietly ignored. (A non-privileged user shall not be able to select an option which is privileged or unsupported.)
- If negotiation of a read-only option is requested, *t_sndudata*( ) may return [TLOOK] or successfully return, but a T_UDERR event occurs to indicate that the datagram was not sent.

  If a privileged option is illegally requested, the option is quietly ignored. (A non-privileged user shall not be able to select an option which is privileged or unsupported.)

If multiple options are submitted to *t_connect*( ), *t_accept*( ) or *t_sndudata*( ) and a read-only option is rejected, the connection or the datagram transmission fails as described. Options that could be successfully negotiated before the erroneous option was processed retain their negotiated values. There is no roll-back mechanism (see also Section 13.4 on page 156).

## 13.4 Option Management of a Transport Endpoint

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

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

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

A transport user can negotiate new option values by calling *t_optmgmt*( ) with the flag T_NEGOTIATE set. The negotiation follows the rules described in Section 13.3 on page 151.

Some options may be modified only in specific XTI states and are read-only in other XTI states. Many options with end-to-end significance, for instance, may not be changed in the state T_DATAXFER, and an attempt to do so will fail (see Section 13.3.6 on page 155). The legal states for each option are specified with its definition.

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

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

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

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

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

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

The negotiation procedure for options at the time of a *t_connect*( ), *t_accept*( ) or *t_sndudata*( ) call always obeys the rules in Section 13.3.3 on page 152 and Section 13.3.4 on page 153, whether the options were explicitly specified during the call or implicitly taken from the internal option buffer.

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

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

## 13.5 Supplements

This section contains supplementary remarks and a short summary.

### 13.5.1 The Option Value T_UNSPEC

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

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

If T_UNSPEC is a legal value for a specific option, it may be used by the user on input, too. It is used to indicate that it is left to the provider to choose an appropriate value. This is especially useful in complex options as ISO throughput, where the option value has an internal structure (see T_TCO_THROUGHPUT in Appendix A on page 265). The transport user may leave some fields unspecified by selecting this value. If the user proposes T_UNSPEC, the transport provider is free to select an appropriate value. This might be the default value, some other explicit value, or T_UNSPEC.

Each option where it is legal to use T_UNSPEC specifies its use as part of its description.

### 13.5.2 The info Argument

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

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

The values in *info→etsdu*, *info→tsdu*, *info→connect* and *info→discon* may be modified as soon as the T_DATAXFER state is entered. Calling *t_optmgmt*( ) need not influence these values (see *t_optmgmt*( ) on page 195).

### 13.5.3 Summary

- The format of an option is defined by a header **struct t_opthdr**, followed by an option value.
- On input, several options can be specified in an input *opt* argument. Each option must begin on a `t_uscalar_t` boundary.
- There are options with end-to-end significance and options without end-to-end significance. On output, the functions *t_listen*( ) and *t_rcvudata*( ) return options with end-to-end significance only. The functions *t_rcvconnect*( ) and *t_rcvuderr*( ) may return options of both categories. On input, options of both categories may be specified with *t_accept*( ) and

*t_sndudata*( ). The functions *t_connect*( ) and *t_optmgmt*( ) can process and return both categories of options.

- A transport endpoint is (logically) associated with an internal option buffer, where the currently effective values are stored. Each successful negotiation of an option modifies this buffer, regardless of whether the call initiating the negotiation succeeds or fails.
- When calling *t_connect*( ), *t_accept*( ) or *t_sndudata*( ), the transport user can choose to submit the currently effective option values by setting the *len* field of the input *opt* argument to zero.
- If a connection is accepted via *t_accept*( ), the explicitly specified option values together with the currently effective option values of *resfd*, not of *fd*, matter in this negotiation step.
- The options returned by *t_rcvuderr*( ) are those negotiated with the outgoing datagram that produced the error. If the error occurred during option negotiation, the returned option might represent some mixture of partly negotiated and not-yet negotiated options.

## 13.6 Portability Aspects

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

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

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

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

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

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

Chapter 14

# XTI Library Functions and Parameters

The function synopses in this Chapter 6, and the function definitions in Appendix E on page 317, conform to ISO C standard (see **referenced documents**).

## 14.1 How to Prepare XTI Applications

In a software development environment, a program, for example **file.c**, that uses XTI functions must be compiled with the XTI Library.

An application may also have a requirement to define some feature test macro(s) as described in Section 1.3 on page 3 (The Compilation Environment) to enable the functionality described in this specification.

The generic XTI structures and constants are all defined in the **<xti.h>** header, which can be found in Appendix E on page 317.

Any protocol-specific header files described in Appendices of this specification and needed by the application must follow the inclusion of **<xti.h>**.

**Note:** When _XOPEN_SOURCE is defined to be less than 500, the definitions from some protocol specific headers are automatically exposed by the inclusion of <xti.h>. The individual appendices for those protocols document the headers for which this applies.

## 14.2 Key for Parameter Arrays

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

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

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

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

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

/ The parameter value is meaningless.

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

## 14.3 Return of TLOOK Error

Many of the XTI functions contained in this chapter return a [TLOOK] error to report the occurrence of an asynchronous event. For these functions a complete list describing the function and the events is provided in Section 12.6 on page 148.

## 14.4 Use of ‘‘struct netbuf’’

Many of the XTI functions have an argument that points to a structure with members of type **struct netbuf**. These are used to pass arbitrary length buffer items such as user data, network addresses and option buffers to and from the XTI interfaces.

**struct netbuf** contains the following members:

```
unsigned int maxlen
unsigned int len
void         *buf
```

For guaranteed portability, the space pointed to by *buf* should be aligned appropriately to the most restrictive alignment of any of the data types it contains.

When a buffer item is being passed to the provider, *buf* and *len* are the address and length in bytes of the data in the user’s buffer. *maxlen* is not used and may take any value.

When a buffer item is being passed to the user, prior to the call the user must set *buf* and *maxlen* to the address and length of the user’s buffer. On return the provider will set *len* to the number of bytes of data placed into the user’s buffer. If the user’s buffer has a non-zero *maxlen* but is not long enough for the buffer item, then the provider will fail the XTI call, setting *t_errno* to TBUFOVFLW and discarding the event being returned to the user. If *maxlen* is zero then the provider will discard the data that would have been returned, and ignore *len* and *buf* (which need not have been a valid address).

**NAME**

t_accept - accept a connection request

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *resfd* | x | / |
| *call→addr.maxlen* | = | = |
| *call→addr.len* | x | = |
| *call→addr.buf* | ? (?) | = |
| *call→opt.maxlen* | = | = |
| *call→opt.len* | x | = |
| *call→opt.buf* | ? (?) | = |
| *call→udata.maxlen* | = | = |
| *call→udata.len* | x | = |
| *call→udata.buf* | ? (?) | = |
| *call→sequence* | x | = |

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

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

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

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

If a different transport endpoint is specified (*resfd!=fd*), then the user may or may not choose to bind the endpoint before the *t_accept*( ) is issued. If the endpoint is not bound prior to the *t_accept*( ), the endpoint must be in the T_UNBND state before the *t_accept*( ) is issued, and the transport provider will automatically bind it to an address that is appropriate for the protocol concerned. 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.

Responding endpoints should be supplied to *t_accept*( ) in the state T_UNBND.

The call to *t_accept*( ) may fail with t_errno set to [TLOOK] if there are indications (for example connect or disconnect) waiting to be received on endpoint *fd*. Applications should be prepared for such a failure.

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

When the user does not indicate any option (*call→opt.len* = 0) the connection shall be accepted with the option values currently set for the responding endpoint *resfd* (see Section 13.3.4 on page 153 and Section 13.4 on page 156).

**CAVEATS**

There may be transport provider-specific restrictions on address binding. See Appendix A on page 265 and Chapter 16 on page 251.

Some transport providers do not differentiate between a connection indication and the connection itself. If the connection has already been established after a successful return of *t_listen*( ), *t_accept*( ) will assign the existing connection to the transport endpoint specified by *resfd* (see Chapter 16 on page 251).

**VALID STATES**

fd: T_INCON
resfd (fd!=resfd): T_IDLE, T_UNBND

**ERRORS**

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

| | |
|---|---|
| [TACCES] | The user does not have permission to accept a connection on the responding transport endpoint or to use the specified options. |
| [TBADADDR] | The specified protocol address was in an incorrect format or contained illegal information. |
| [TBADDATA] | The amount of user data specified was not within the bounds allowed by the transport provider. |
| [TBADF] | The file descriptor *fd* or *resfd* does not refer to a transport endpoint. |
| [TBADOPT] | The specified options were in an incorrect format or contained illegal information. |
| [TBADSEQ] | Either an invalid sequence number was specified, or a valid sequence number was specified but the connection request was aborted by the peer. In the latter case, its T_DISCONNECT event will be received on the listening endpoint. |
| [TINDOUT] | The function was called with *fd==resfd* but there are outstanding connection indications on the endpoint. Those other connection indications must be handled either by rejecting them via *t_snddis*(3) or accepting them on a different endpoint via *t_accept*(3). |
| [TLOOK] | An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* or *resfd* is not in one of the states in which a call to this function is valid. |

[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 error *(t_errno)*.

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

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

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

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

**RETURN VALUE**

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

**SEE ALSO**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_alloc - allocate a library structure

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *struct_type* | x | / |
| *fields* | x | / |

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

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

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

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

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

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

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

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

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

For each relevant field specified in *fields*, *t_alloc*( ) will allocate memory for the buffer associated with the field, and initialise the *len* field to zero and the *buf* pointer and *maxlen* field accordingly. Irrelevant or unknown values passed in fields are ignored. Since the length of the buffer allocated will be based on the same size information that is returned to the user on a call to *t_open*( ) and *t_getinfo*( ), *fd* must refer to the transport endpoint through which the newly allocated structure will be passed. In the case where a T_INFO structure is to be allocated, *fd* may be set to any value. In this way the appropriate size information can be accessed. If the size value associated with any specified field is T_INVALID (see *t_open*( ) or *t_getinfo*( )), *t_alloc*( ) will be unable to determine the size of the buffer to allocate and will fail, setting *t_errno* to [TSYSERR] and *errno* to [EINVAL]. If the size value associated with any specified field is T_INFINITE (see

*t_open*( ) or *t_getinfo*( )), then the behaviour of *t_alloc*( ) is implementation-defined. For any field not specified in *fields*, *buf* will be set to the null pointer and *len* and *maxlen* will be set to zero.

The pointer returned if the allocation succeeds is suitably aligned so that it can be assigned to a pointer to any type of object and then used to access such an object or array of such objects in the space allocated. The pointer references to space allocations embedded in *struct netbuf* fields (pointed by the ,I buf pointers) are also aligned in the same way.

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

**VALID STATES**

ALL - apart from T_UNINIT

**ERRORS**

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

[TBADF] **struct_type** is other than T_INFO and the specified file descriptor does not refer to a transport endpoint.

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

[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 error *(t_errno)*.

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

**RETURN VALUE**

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

**SEE ALSO**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_bind - bind an address to a transport endpoint

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *req→addr.maxlen* | = | = |
| *req→addr.len* | x≥0 | = |
| *req→addr.buf* | x (x) | = |
| *req→qlen* | x≥0 | = |
| *ret→addr.maxlen* | x | = |
| *ret→addr.len* | / | x |
| *ret→addr.buf* | ? | (?) |
| *ret→qlen* | / | x≥0 |

*ret→addr.buf* (the pointer itself, not the buffer it points to) is also unchanged.

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

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

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

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

The parameter *req* is used to request that an address, represented by the **netbuf** structure, be bound to the given transport endpoint. The parameter *len* specifies the number of bytes in the address, and *buf* points to the address buffer. The parameter *maxlen* has no meaning for the *req* argument. On return, *ret* contains an encoding for the address that the transport provider actually bound to the transport endpoint; if an address was specified in *req*, this will be an encoding of the same address. 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* equals zero, no address is returned. If *maxlen* is greater than zero and less than the length of the address, *t_bind*( ) fails with *t_errno* set to [TBUFOVFLW].

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

The parameter *req* may be a null pointer if the user does not wish to specify an address to be bound. Here, the value of *qlen* is assumed to be zero, and the transport provider will assign an address to the transport endpoint. Similarly, *ret* may be a null pointer if the user does not care

what address was bound by the provider and is not interested in the negotiated value of *qlen*. It is valid to set *req* and *ret* to the null pointer for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

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

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

If *fd* refers to connectionless mode service, this function allows for more than one transport endpoint to be associated with a protocol address, where the underlying transport provider supports this capability (often in conjunction with value of a protocol-specific option). If a user attempts to bind a second transport endpoint to an already bound protocol address when such capability is not supported for a transport provider, *t_bind*( ) will return –1 and set *t_errno* to [TADDRBUSY].

**VALID STATES**

T_UNBND

**ERRORS**

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

| | |
|---|---|
| [TACCES] | The user does not have permission to use the specified address. |
| [TADDRBUSY] | The requested address is in use. |
| [TBADADDR] | The specified protocol address was in an incorrect format or contained illegal information. |
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TBUFOVFLW] | The number of bytes allowed for an incoming argument *(maxlen)* is greater than 0 but not sufficient to store the value of that argument. The provider's state will change to T_IDLE and the information to be returned |

in *ret* will be discarded.

[TOUTSTATE] The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

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

[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 error *(t_errno)*.

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

**RETURN VALUE**

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

**SEE ALSO**

*t_accept*( ), *t_alloc*( ), *t_close*( ), *t_connect*( ), *t_unbind*( ).

**CAVEATS**

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

An implementation need not allow an application explicitly to bind more than one communications endpoint to a single protocol address, while permitting more than one connection to be accepted to the same protocol address. That means that although an attempt to bind a communications endpoint to some address with *qlen=0* might be rejected with [TADDRBUSY], the user may nevertheless use this (unbound) endpoint as a responding endpoint in a call to *t_accept*( ). To become independent of such implementation differences, the user should supply unbound responding endpoints to *t_accept*( ).

The local address bound to an endpoint may change as result of a *t_accept*( ) or *t_connect*( ) call. Such changes are not necessarily reversed when the connection is released.

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_close - close a transport endpoint

**SYNOPSIS**

```
#include <xti.h>

int t_close(int fd);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |

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

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

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

For advice on how to write protocol-independent applications, see Section B.4 on page 283. For information on protocol-specific behaviour of *t_close*( ), see the XTI Appendix for the relevant transport provider.

**VALID STATES**

ALL - apart from T_UNINIT

**ERRORS**

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

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

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

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

**RETURN VALUE**

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

**SEE ALSO**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_connect - establish a connection with another transport user

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *sndcall→addr.maxlen* | = | = |
| *sndcall→addr.len* | x | = |
| *sndcall→addr.buf* | x (x) | = |
| *sndcall→opt.maxlen* | = | = |
| *sndcall→opt.len* | x | = |
| *sndcall→opt.buf* | x (x) | = |
| *sndcall→udata.maxlen* | = | = |
| *sndcall→udata.len* | x | = |
| *sndcall→udata.buf* | ? (?) | = |
| *sndcall→sequence* | = | = |
| *rcvcall→addr.maxlen* | x | / |
| *rcvcall→addr.len* | / | x |
| *rcvcall→addr.buf* | ? | (?) |
| *rcvcall→opt.maxlen* | x | = |
| *rcvcall→opt.len* | / | x |
| *rcvcall→opt.buf* | ? | (?) |
| *rcvcall→udata.maxlen* | x | = |
| *rcvcall→udata.len* | / | x |
| *rcvcall→udata.buf* | ? | (?) |
| *rcvcall→sequence* | = | = |

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

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

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

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

On return, in *rcvcall*, *addr* contains the protocol address associated with the responding transport endpoint, *opt* represents any protocol-specific information associated with the connection, *udata* points to optional user data that may be returned by the destination transport user during

connection establishment, and *sequence* has no meaning for this function.

The *opt* argument permits users to define the options that may be passed to the transport provider. These options are specific to underlying protocol of the transport provider or XTI interface and are described in Appendix E (for the XTI interface) and other protocol-specific appendices which are part of this specification. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider uses the option values currently set for the communications endpoint.

If used, *sndcall→opt.buf* must point to a buffer with the corresponding options, and *sndcall→opt.len* must specify its length. The *maxlen* and *buf* fields of the **netbuf** structure pointed by *rcvcall→addr* and *rcvcall→opt* must be set before the call.

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

On return, the *addr*, *opt* and *udata* fields of *rcvcall* will be updated to reflect values associated with the connection. Thus, the *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *maxlen* can be set to zero, in which case no information to this specific argument is given to the user on the return from *t_connect*( ). If maxlen is greater than zero and less than the length of the value, *t_connect*( ) fails with *t_errno* set to [TBUFOVFLW]. If *rcvcall* is set to NULL, no information at all is returned.

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

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

**VALID STATES**

T_IDLE

**ERRORS**

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

| | |
|---|---|
| [TACCES] | The user does not have permission to use the specified address or options. |
| [TADDRBUSY] | This transport provider does not support multiple connections with the same local and remote addresses. This error indicates that a connection already exists. |
| [TBADADDR] | The specified protocol address was in an incorrect format or contained illegal information. |

| | |
|---|---|
| [TBADDATA] | The amount of user data specified was not within the bounds allowed by the transport provider. |
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TBADOPT] | The specified protocol options were in an incorrect format or contained illegal information. |
| [TBUFOVFLW] | The number of bytes allocated for an incoming argument *(maxlen)* is greater than 0 but not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the information to be returned in *rcvcall* is discarded. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNODATA] | O_NONBLOCK was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_error - produce error message

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *errmsg* | x | = |

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

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

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

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

**EXAMPLE**

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

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

The diagnostic message to be printed would look like:

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

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

**VALID STATES**

All - apart from T_UNINIT

**ERRORS**

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

**RETURN VALUE**

Upon completion, a value of 0 is returned.

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_errno - XTI error return value

**SYNOPSIS**

```
#include <xti.h>
```

**DESCRIPTION**

*t_errno* is used by XTI functions to return error values.

XTI functions provide an error number in *t_errno* which has type *int* and is defined in **<xti.h>**. The value of *t_errno* will be defined only after a call to a XTI function for which it is explicitly stated to be set and until it is changed by the next XTI function call. The value of *t_errno* should only be examined when it is indicated to be valid by a function's return value. Programs should obtain the definition of *t_errno* by the inclusion of **<xti.h>**. The practice of defining *t_errno* in program as *extern*in *t_errno* is obsolescent. No XTI function sets *t_errno* to 0 to indicate an error.

It is unspecified whether *t_errno* is a macro or an identifier with external linkage. It represents a modifiable **lvalue** of type *int*. If a macro definition is suppressed in order to access an actual object or a program defines an identifier with name *t_errno*, the behavior is undefined.

The symbolic values stored in *t_errno* by an XTI function are defined in the **ERRORS** sections in all relevant XTI function definition pages.

**NAME**

t_free - free a library structure

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *ptr* | x | / |
| *struct_type* | x | / |

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

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

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

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

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

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

**VALID STATES**

ALL - apart from T_UNINIT

**ERRORS**

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

[TNOSTRUCTYPE] Unsupported *struct_type* requested.

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

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

**RETURN VALUE**

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

**SEE ALSO**

*t_alloc*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_getinfo - get protocol-specific service information

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *info→addr* | / | x |
| *info→options* | / | x |
| *info→tsdu* | / | x |
| *info→etsdu* | / | x |
| *info→connect* | / | x |
| *info→discon* | / | x |
| *info→servtype* | / | x |
| *info→flags* | / | x |

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

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

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

The values of the fields have the following meanings:

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

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

*tsdu* A value greater than zero specifies the maximum size in octets of a transport service data unit (TSDU); a value of T_NULL (zero) specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a datastream with no logical boundaries preserved across a connection; a value of T_INFINITE (–1) specifies that there is no limit on the size in octets of a TSDU; and a value

of T_INVALID (–2) specifies that the transfer of normal data is not supported by the transport provider.

*etsdu* A value greater than zero specifies the maximum size in octets of an expedited transport service data unit (ETSDU); a value of T_NULL (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 T_INFINITE (–1) specifies that there is no limit on the size (in octets) of an ETSDU; and a value of T_INVALID (–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 265 and Chapter 16 on page 251).

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

*discon* If the T_ORDRELDATA bit in flags is clear, a value greater than zero specifies the maximum number of octets that may be associated with the *t_snddis*( ) and *t_rcvdis*( ) functions, and a value of T_INVALID (–2) specifies that the transport provider does not allow data to be sent with the abortive release functions. If the T_ORDRELDATA bit is set in flags, a value greater than zero specifies the maximum number of octets that may be associated with the *t_sndreldata*( ), *t_rcvreldata*( ), *t_snddis*( ) and *t_rcvdis*( ) functions.

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

*flags* This is a bit field used to specify other information about the communications provider. If the T_ORDRELDATA bit is set, the communications provider supports sending user data with an orderly release. 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 265 for a discussion of the separate issue of zero-length fragments within a TSDU.

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

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

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

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

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

*discon*.

**VALID STATES**

ALL - apart from T_UNINIT

**ERRORS**

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

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

**RETURN VALUE**

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

**SEE ALSO**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_getprotaddr - get the protocol addresses

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *boundaddr→addr.maxlen* | x | = |
| *boundaddr→addr.len* | / | x |
| *boundaddr→addr.buf* | ? | (?) |
| *boundaddr→qlen* | = | = |
| *peeraddr→addr.maxlen* | x | = |
| *peeraddr→addr.len* | / | x |
| *peeraddr→addr.buf* | ? | (?) |
| *peeraddr→qlen* | = | = |

The *t_getprotaddr*( ) function returns local and remote protocol addresses currently associated with the transport endpoint specified by *fd*. In *boundaddr* and *peeraddr* the user specifies *maxlen*, which is the maximum size (in bytes) 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, T_INREL, T_OUTCON or T_OUTREL states, zero is returned in the *len* field of *peeraddr*. If the *maxlen* field of *boundaddr* or *peeraddr* is set to zero, no address is returned.

**VALID STATES**

ALL - apart from T_UNINIT

**ERRORS**

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

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

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

[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 error *(t_errno)*.

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

**RETURN VALUE**

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

**SEE ALSO**

*t_bind*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_getstate - get the current state

**SYNOPSIS**

```
#include <xti.h>

int t_getstate(int fd);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |

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

**VALID STATES**

ALL - apart from T_UNINIT

**ERRORS**

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

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

[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 error *(t_errno)*.

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

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

**RETURN VALUE**

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

T_UNBND Unbound.

T_IDLE Idle.

T_OUTCON Outgoing connection pending.

T_INCON Incoming connection pending.

T_DATAXFER Data transfer.

T_OUTREL Outgoing direction orderly release sent.

T_INREL Incoming direction orderly release received.

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

**SEE ALSO**

*t_open*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_listen - listen for a connection indication

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *call→addr.maxlen* | x | = |
| *call→addr.len* | / | x |
| *call→addr.buf* | ? | (?) |
| *call→opt.maxlen* | x | = |
| *call→opt.len* | / | x |
| *call→opt.buf* | ? | (?) |
| *call→udata.maxlen* | x | = |
| *call→udata.len* | / | x |
| *call→udata.buf* | ? | (?) |
| *call→sequence* | / | x |

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

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

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

Since this function returns values for the *addr*, *opt* and *udata* fields of *call*, the *maxlen* field of each must be set before issuing the *t_listen*( ) to indicate the maximum size of the buffer for each. If the *maxlen* field of *call→addr*, *call→opt* or *call→udata* is set to zero, no information is returned for this parameter.

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

**VALID STATES**

T_IDLE, T_INCON

**ERRORS**

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

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

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

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

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

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

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

[TOUTSTATE] The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

[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 error *(t_errno)*.

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

Note that a subsequent call to *t_listen*( ) may block until another incoming connection indication is available. This can only occur if at least one of the outstanding connection indications becomes no longer outstanding, for example through a call to *t_accept*( ).

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

**CAVEATS**

Some transport providers do not differentiate between a connection indication and the connection itself. If this is the case, a successful return of *t_listen*( ) indicates an existing connection (see Chapter 16 on page 251).

**RETURN VALUE**

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

**SEE ALSO**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_look - look at the current event on a transport endpoint

**SYNOPSIS**

```
#include <xti.h>

int t_look(int fd);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |

This function returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is calling functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, [TLOOK], on the current or next function to be executed. Details on events which cause functions to fail [TLOOK] may be found in Section 12.6 on page 148.

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

**VALID STATES**

ALL - apart from T_UNINIT

**ERRORS**

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

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

[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 error *(t_errno)*.

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

**RETURN VALUE**

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

T_LISTEN Connection indication received.

T_CONNECT Connect confirmation received.

T_DATA Normal data received.

T_EXDATA Expedited data received.

T_DISCONNECT Disconnection received.

T_UDERR Datagram error indication.

T_ORDREL Orderly release indication.

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

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

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

**SEE ALSO**

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

**APPLICATION USAGE**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_open - establish a transport endpoint

**SYNOPSIS**

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

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *name* | x | = |
| *oflag* | x | = |
| *info→addr* | / | x |
| *info→options* | / | x |
| *info→tsdu* | / | x |
| *info→etsdu* | / | x |
| *info→connect* | / | x |
| *info→discon* | / | x |
| *info→servtype* | / | x |
| *info→flags* | / | x |

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

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

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

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

The values of the fields have the following meanings:

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

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

*tsdu* A value greater than zero (T_NULL) specifies the maximum size of a transport service data unit (TSDU); a value of zero (T_NULL) 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 (T_INFINITE) specifies that there is no limit to the size of a TSDU; and a value of –2 (T_INVALID) specifies that the transfer of normal data is not supported by the transport provider.

*etsdu* A value greater than zero (T_NULL) specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero (T_NULL) 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 (T_INFINITE) specifies that there is no limit on the size of an ETSDU; and a value of –2 (T_INVALID) 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 265 and Chapter 16 on page 251).

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

*discon* If the T_ORDRELDATA bit in flags is clear, a value greater than zero (T_NULL) specifies the maximum amount of data that may be associated with the *t_snddis*( ) and *t_rcvdis*( ) functions, and a value of –2 (T_INVALID) specifies that the transport provider does not allow data to be sent with the abortive release functions. If the T_ORDRELDATA bit is set in flags, a value greater than zero (T_NULL) specifies the maximum number of octets that may be associated with the *t_sndreldata*( ), *t_rcvreldata*( ), *t_snddis*( ) and *t_rcvdis*( ) functions.

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

*flags* This is a bit field used to specify other information about the communications provider. If the T_ORDRELDATA bit is set, the communications provider supports user data to be sent with an orderly release. 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 265 for a discussion of the separate issue of zero-length fragments within a TSDU.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the

*t_alloc*( ) function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.

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

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

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

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

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

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

**VALID STATES**

T_UNINIT

**ERRORS**

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

[TBADFLAG] An invalid flag is specified.

[TBADNAME] Invalid transport provider name.

[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 error *(t_errno)*.

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

**RETURN VALUES**

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

**SEE ALSO**

*open*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_optmgmt - manage options for a transport endpoint

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *req→opt.maxlen* | = | = |
| *req→opt.len* | x | = |
| *req→opt.buf* | x (x) | = |
| *req→flags* | x | = |
| *ret→opt.maxlen* | x | = |
| *ret→opt.len* | / | x |
| *ret→opt.buf* | ? | (?) |
| *ret→flags* | / | x |

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

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

```
struct      netbuf opt;
t_scalar_t  flags;
```

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

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

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

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

Several options can be concatenated. The option user has, however to ensure that each options header and value part starts at a boundary appropriate for the architecture-specific alignment rules. The macros T_OPT_FIRSTHDR(nbp), T_OPT_NEXTHDR(nbp, tohp), T_OPT_DATA(tohp) are provided for that purpose.

T_OPT_DATA(tohp) If argument is a pointer to a **t_opthdr** structure, this macro returns an unsigned character pointer to the data associated with the **t_opthdr**.

T_OPT_NEXTHDR(nbp, tohp)
If the first argument is a pointer to a netbuf structure associated with an option buffer and second argument is a pointer to a **t_opthdr** structure within that option buffer, this macro returns a pointer to the next **t_opthdr** structure or a null pointer if this **t_opthdr** is the last **t_opthdr** in the option buffer. In this case, the space remaining in the option buffer is none or too small to accomodate a **t_opthdr**.

T_OPT_FIRSTHDR(nbp)
If the argument is a pointer to a **netbuf** structure associated with an option buffer, this macro returns the pointer to the first **t_opthdr** structure in the associated option buffer, or a null pointer if there is no option buffer associated with this **netbuf** or if it is not possible or the associated option buffer is too small to accommodate even the first aligned option header.

T_OPT_FIRSTHDR(nbp) is useful for finding an appropriately aligned start of the option buffer. T_OPT_NEXTHDR(nbp, tohp) is useful for moving to the start of the next appropriately aligned option in the option buffer. T_OPT_DATA(tohp) is useful for finding the start of the data part in the option buffer where the contents of its values start on an appropriately aligned boundary.

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

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

Chapter 13 contains a detailed description about the use of options and should be read before using this function.

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

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

The overall result of the negotiation is returned in *ret→flags*.

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

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

T_CHECK

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

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

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

The overall result of the option checks is returned in *ret→flags*. This field contains the worst single result of the option checks, whereby the rating is the same as for T_NEGOTIATE.

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

T_DEFAULT

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

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

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

T_CURRENT

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

The *status* field returned is T_NOTSUPPORT if the protocol level does not support this option or the transport user illegally requested a privileged option, T_READONLY if the option is read-only, and set to T_SUCCESS in all other cases. The overall result of the request is returned in

*ret→flags*. This field contains the worst single result, whereby the rating is the same as for T_NEGOTIATE.

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

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

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

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

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

**XTI-LEVEL OPTIONS**

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

The subsequent options do not have end-to-end significance (see Chapter 13). They may be negotiated in all XTI states except T_UNINIT.

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

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

**Table 14-1** XTI-level Options

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

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

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

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

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

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

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

Legal values for the field *l_onoff* are:

T_NO switch option off
T_YES activate option

The value *l_onoff* is an absolute requirement.

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

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

the lower limit are negotiated to the lower limit.

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

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

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

Legal values are all positive numbers.

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

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

Legal values are all positive numbers.

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

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

Legal values are all positive numbers.

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

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

Legal values are all positive numbers.

**VALID STATES**

ALL - apart from T_UNINIT

**ERRORS**

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

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

[TBADFLAG] An invalid flag was specified.

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

| | |
|---|---|
| [TBUFOVFLW] | The number of bytes allowed for an incoming argument *(maxlen)* is greater than 0 but not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded. |
| [TNOTSUPPORT] | This action is not supported by the transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

*t_accept*( ), *t_alloc*( ), *t_connect*( ), *t_getinfo*( ), *t_listen*( ), *t_open*( ), *t_rcvconnect*( ), Chapter 13.

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_rcv - receive data or expedited data sent over a connection

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

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

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

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

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

On return, the data is expedited if T_EXPEDITED is set in flags. If T_MORE is also set, it indicates that the number of expedited bytes exceeded nbytes, a signal has interrupted the call, or that an entire ETSDU was not available (only for transport protocols that support fragmentation of ETSDUs). The rest of the ETSDU will be returned by subsequent calls to *t_rcv*( ) which will return with T_EXPEDITED set in flags. The end of the ETSDU is identified by the return of a *t_rcv*( ) call with T_EXPEDITED set and T_MORE cleared. If the entire ETSDU is not available it is possible for normal data fragments to be returned between the initial and final fragments of an ETSDU.

If a signal arrives, *t_rcv*( ) returns, giving the user any data currently available. If no data is available, *t_rcv*( ) returns –1, sets *t_errno* to [TSYSERR] and *errno* to [EINTR]. If some data is available, *t_rcv*( ) returns the number of bytes received and T_MORE is set in flags.

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

**VALID STATES**

T_DATAXFER, T_OUTREL

**ERRORS**

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

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNODATA] | O_NONBLOCK was set, but no data is currently available from the transport provider. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_rcvconnect - receive the confirmation from a connection request

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *call→addr.maxlen* | x | = |
| *call→addr.len* | / | x |
| *call→addr.buf* | ? | (?) |
| *call→opt.maxlen* | x | = |
| *call→opt.len* | / | x |
| *call→opt.buf* | ? | (?) |
| *call→udata.maxlen* | x | = |
| *call→udata.len* | / | x |
| *call→udata.buf* | ? | (?) |
| *call→sequence* | = | = |

This function enables a calling transport user to determine the status of a previously sent connection request and is used in conjunction with *t_connect*( ) to establish a connection in asynchronous mode, and to complete a synchronous *t_connect*( ) call that was interrupted by a signal. The connection will be established on successful completion of this function.

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

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

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

The *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *maxlen* can be set to zero, in which case no information to this specific argument is given to the user on the return from *t_rcvconnect*( ). If *call* is set to NULL, no information at all is returned. By default, *t_rcvconnect*( ) executes in synchronous mode and waits for the connection to be established before returning. On return, the *addr*, *opt* and *udata* fields reflect values associated with the connection.

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

**VALID STATES**

T_OUTCON

**ERRORS**

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

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TBUFOVFLW] | The number of bytes allocated for an incoming argument *(maxlen)* is greater than 0 but not sufficient to store the value of that argument, and the connection information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to T_DATAXFER. |
| [TLOOK] | An asynchronous event has occurred on this transport connection and requires immediate attention. |
| [TNODATA] | O_NONBLOCK was set, but a connection confirmation has not yet arrived. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_rcvdis - retrieve information from disconnection

**SYNOPSIS**

```
#include <xti.h>

int t_rcvdis(int fd, struct t_discon *discon);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *discon→udata.maxlen* | x | = |
| *discon→udata.len* | / | x |
| *discon→udata.buf* | ? | (?) |
| *discon→reason* | / | x |
| *discon→sequence* | / | ? |

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

```
struct netbuf udata;
int reason;
int sequence;
```

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

The *maxlen* field of *udata* may be set to zero, if the user does not care about incoming data. If, in addition, the user does not need to know the value of *reason* or *sequence*, *discon* may be set to NULL and any user data associated with the disconnection indication shall be discarded. However, if a user has retrieved more than one outstanding connection indication (via *t_listen*( )) and *discon* is a null pointer, the user will be unable to identify with which connection indication the disconnection is associated.

**VALID STATES**

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

**ERRORS**

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

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

[TBUFOVFLW] The number of bytes allocated for incoming data *(maxlen)* is greater than 0 but not sufficient to store the data. If *fd* is a passive endpoint with *ocnt* > 1, it remains in state T_INCON; otherwise, the endpoint state is set to T_IDLE.

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

| | |
|---|---|
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

*t_alloc*( ), *t_connect*( ), *t_listen*( ), *t_open*( ), *t_snddis*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_rcvrel - acknowledge receipt of an orderly release indication

**SYNOPSIS**

```
#include <xti.h>

int t_rcvrel(int fd);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |

This function is used to receive an orderly release indication for the incoming direction of data transfer. The argument *fd* identifies the local transport endpoint where the connection exists. After receipt of this indication, the user may not attempt to receive more data via *t_rcv*( ) or *t_rcvv*( ). Such an attempt will fail with *t_error* set to [TOUTSTATE]. However, the user may continue to send data over the connection if *t_sndrel*( ) has not been called by the user. This function is an optional service of the transport provider, and is only supported if the transport provider returned service type T_COTS_ORD on *t_open*( ) or *t_getinfo*( ). Any user data that may be associated with the orderly release indication is discarded when *t_rcvrel*( ) is called.

**VALID STATES**

T_DATAXFER,T_OUTREL

**ERRORS**

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

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

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

[TNOREL] No orderly release indication currently exists on the specified transport endpoint.

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

[TOUTSTATE] The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

[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 error *(t_errno)*.

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

**RETURN VALUE**

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

**SEE ALSO**

*t_getinfo*( ), *t_open*( ), *t_sndrel*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_rcvreldata - receive an orderly release indication or confirmation containing user data

**SYNOPSIS**

```
#include <xti.h>

int t_rcvreldata(int fd, struct t_discon *discon);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *discon→udata.maxlen* | x | = |
| *discon→udata.len* | / | x |
| *discon→udata.buf* | ? | (?) |
| *discon→reason* | / | x |
| *discon→sequence* | / | = |

This function is used to receive an orderly release indication for the incoming direction of data transfer and to retrieve any user data sent with the release. The argument *fd* identifies the local transport endpoint where the connection exists, and *discon* points to a **t_discon** structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

After receipt of this indication, the user may not attempt to receive more data via *t_rcv*( ) or *t_rcvv*( ). Such an attempt will fail with *t_error* set to [TOUTSTATE]. However, the user may continue to send data over the connection if *t_sndrel*( ) or *t_sndreldata*( ) has not been called by the user.

The field *reason* specifies the reason for the disconnection through a protocol-dependent *reason code*, and *udata* identifies any user data that was sent with the disconnection; the field *sequence* is not used.

If a user does not care if there is incoming data and does not need to know the value of *reason*, *discon* may be a null pointer, and any user data associated with the disconnection will be discarded.

If *discon→udata.maxlen* is greater than zero and less than the length of the value, *t_rcvreldata*( ) fails with *t_errno* set to [TBUFOVFLW].

This function is an optional service of the transport provider, only supported by providers of service type T_COTS_ORD. The flag T_ORDRELDATA in the *info→flag* field returned by *t_open*( ) or *t_getinfo*( ) indicates that the provider supports orderly release user data; when the flag is not set, this function behaves as *t_rcvrel*( ) and no user data is returned.

This function may not be available on all systems.

**VALID STATES**

T_DATAXFER, T_OUTREL

**ERRORS**

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

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

| | |
|---|---|
| [TBUFOVFLW] | The number of bytes allocated for incoming data (maxlen) is greater than 0 but not sufficient to store the data, and the disconnection information to be returned in *discon* will be discarded. The provider state, as seen by the user, will be changed as if the data was successfully retrieved. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNOREL] | No orderly release indication currently exists on the specified transport endpoint. |
| [TNOTSUPPORT] | Orderly release is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

*t_getinfo*( ), *t_open*( ), *t_sndreldata*( ), *t_rcvrel*( ), *t_sndrel*( ).

**NAME**

t_rcvudata - receive a data unit

**SYNOPSIS**

```
#include <xti.h>

int t_rcvudata(int fd, struct t_unitdata *unitdata, int *flags);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *unitdata→addr.maxlen* | x | = |
| *unitdata→addr.len* | / | x |
| *unitdata→addr.buf* | ? | (?) |
| *unitdata→opt.maxlen* | x | = |
| *unitdata→opt.len* | / | x |
| *unitdata→opt.buf* | ? | (?) |
| *unitdata→udata.maxlen* | x | = |
| *unitdata→udata.len* | / | x |
| *unitdata→udata.buf* | ? | (?) |
| *flags* | / | x |

This function is used in connectionless-mode to receive a data unit from another transport user. The argument *fd* identifies the local transport endpoint through which data will be received, *unitdata* holds information associated with the received data unit, *iovcount* contains the number of non-contiguous udata buffers and is greater than zero and limited to an implementation-defined value given by T_IOV_MAX which is at least 16, and *flags* is set on return to indicate that the complete data unit was not received. If the limit on *iovcount* is exceeded or iovcount is zero, the function fails with [TBADDATA]. The argument *unitdata* points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The *maxlen* field of *addr*, *opt* and *udata* must be set before calling this function to indicate the maximum size of the buffer for each. If the *maxlen* field of *addr* or *opt* is set to zero, no information is returned in the *buf* field of this parameter.

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

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

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

If the call is interrupted, *t_rcvudata*() will return [EINTR] and no datagrams will have been removed from the endpoint.

**VALID STATES**

T_IDLE

**ERRORS**

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

| | |
|---|---|
| [TBADDATA] | *iovcount* is zero or greater than T_IOV_MAX. |
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TBUFOVFLW] | The number of bytes allocated for the incoming protocol address or options *(maxlen)* is greater than 0 but not sufficient to store the information. The unit data information to be returned in *unitdata* will be discarded. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNODATA] | O_NONBLOCK was set, but no data units are currently available from the transport provider. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_rcvuderr - receive a unit data error indication

**SYNOPSIS**

```
#include <xti.h>

int t_rcvuderr(int fd, struct t_uderr *uderr);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *uderr→addr.maxlen* | x | = |
| *uderr→addr.len* | / | x |
| *uderr→addr.buf* | ? | (?) |
| *uderr→opt.maxlen* | x | = |
| *uderr→opt.len* | / | x |
| *uderr→opt.buf* | ? | (?) |
| *uderr→error* | / | x |

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

```
struct netbuf addr;
struct netbuf opt;
t_scalar_t error;
```

The *maxlen* field of *addr* and *opt* must be set before calling this function to indicate the maximum size of the buffer for each. If this field is set to zero for *addr* or *opt*, no information is returned in the *buf* field of this parameter.

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

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

**VALID STATES**

T_IDLE

**ERRORS**

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

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

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

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

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

[TOUTSTATE] The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

[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 error *(t_errno)*.

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

**RETURN VALUE**

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

**SEE ALSO**

*t_rcvudata*( ), *t_sndudata*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_rcvv - receive data or expedited data sent over a connection and put the data into one or more non-contiguous buffers

**SYNOPSIS**

```
#include <xti.h>

int t_rcvv(int fd, struct t_iovec *iov, unsigned int iovcount,
    int *flags);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *iov* | x / | |
| *iovcount* | x | / |
| *iov[0].iov_base* | x( / ) | =(x) |
| *iov[0].iov_len* | x | = |
| . . . . | | |
| *iov[iovcount-1].iov_base* | x( / ) | =(x) |
| *iov[iovcount-1].iov_len* | x | = |

This function receives either normal or expedited data. The argument *fd* identifies the local transport endpoint through which data will arrive, *iov* points to an array of buffer address / buffer size pairs (*iov_base*, *iov_len*). The *t_rcvv*( ) function receives data into the buffers specified by *iov[0].iov_base*, *iov[1].iov_base*, through *iov[iovcount-1].iov_base*, always filling one buffer before proceding to the next.

**Note:** The limit on the total number of bytes available in all buffers passed (that is, *iov(0).iov_len* + . . + *iov(iovcount-1).iov_len)* may be constrained by implementation limits. If no other constraint applies, it will be limited by [INT_MAX]. In practice, the availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter / gather functions.

The argument *iovcount* contains the number of buffers which is limited to T_IOV_MAX (an implementation-defined value of at least 16). If the limit is exceeded, the function will fail with [TBADDATA].

The argument flags may be set on return from *t_rcvv*( ) and specifies optional flags as described below.

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

On return from the call, if T_MORE is set in flags, this indicates that there is more data, and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple *t_rcvv*( ) or *t_rcv*( ) calls. In the asynchronous mode, or under unusual conditions (for example, the arrival of a signal or T_EXDATA event), the T_MORE flag may be set on return from the *t_rcvv*( ) call even when the number of bytes received is less than the total size of all the receive buffers. Each *t_rcvv*( ) with the T_MORE flag set indicates that another *t_rcvv*( ) must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a *t_rcvv*( ) call with the T_MORE flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from *t_open*( ) or *t_getinfo*( ), the T_MORE flag is not meaningful and should be ignored. If the amount of buffer space passed in *iov* is greater than zero on the call to *t_rcvv*( ), then *t_rcvv*( ) will return 0

only if the end of a TSDU is being returned to the user.

On return, the data is expedited if T_EXPEDITED is set in flags. If T_MORE is also set, it indicates that the number of expedited bytes exceeded nbytes, a signal has interrupted the call, or that an entire ETSDU was not available (only for transport protocols that support fragmentation of ETSDUs). The rest of the ETSDU will be returned by subsequent calls to *t_rcvv*( ) which will return with T_EXPEDITED set in flags. The end of the ETSDU is identified by the return of a *t_rcvv*( ) call with T_EXPEDITED set and T_MORE cleared. If the entire ETSDU is not available it is possible for normal data fragments to be returned between the initial and final fragments of an ETSDU.

If a signal arrives, *t_rcvv*( ) returns, giving the user any data currently available. If no data is available, *t_rcvv*( ) returns –1, sets *t_errno* to [TSYSERR] and *errno* to [EINTR]. If some data is available, *t_rcvv*( ) returns the number of bytes received and T_MORE is set in flags.

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

**VALID STATES**

T_DATAXFER, T_OUTREL

**ERRORS**

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

| | |
|---|---|
| [TBADDATA] | *iovcount* is zero or greater than T_IOV_MAX. |
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNODATA] | O_NONBLOCK was set, but no data is currently available from the transport provider. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

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

**NAME**

t_rcvvudata — receive a data unit into one or more noncontiguous buffers

**SYNOPSIS**

```
#include <xti.h>

int t_rcvvudata(int fd, struct t_unitdata *unitdata,
    struct t_iovec *iov, unsigned int iovcount, int *flags);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *unitdata->addr.maxlen* | x | = |
| *unitdata->addr.len* | / | x |
| *unitdata->addr.buf* | ?( / ) | =( / ) |
| *unitdata->opt.maxlen* | x | = |
| *unitdata->opt.len* | / | x |
| *unitdata->opt.buf* | ?( / ) | =(?) |
| *unitdata->udata.maxlen* | / | = |
| *unitdata->udata.len* | / | = |
| *unitdata->udata.buf* | / | = |
| *iov[0].iov_base* | x | =(x) |
| *iov[0].iov_len* | x | = |
| . . . . | | |
| *iov[iovcount-1].iov_base* | x( / ) | =(x) |
| *iov[iovcount-1].iov_len* | x | = |
| *iovcount* | x | / |
| *flags* | / | / |

This function is used in connectionless mode to receive a data unit from another transport user. The argument *fd* identifies the local transport endpoint through which data will be received, *unitdata* holds information associated with the received data unit, *iovcount* contains the number of non-contiguous udata buffers which is limited to T_IOV_MAX (an implementation-defined value of at least 16), and *flags* is set on return to indicate that the complete data unit was not received. If the limit on *iovcount* is exceeded, the function fails with [TBADDATA]. The argument *unitdata* points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The *maxlen* field of *addr* and *opt* must be set before calling this function to indicate the maximum size of the buffer for each. The *udata* field of **t_unitdata** is not used. The *iov_len* and *iov_base* fields of *iov[0]* through *iov[iovcount-1]* must be set before calling *t_rcvvudata*( ) to define the buffer where the userdata will be placed. If the maxlen field of *addr* or *opt* is set to zero then no information is returned in the *buf* field for this parameter.

On return from this call, *addr* specifies the protocol address of the sending user, *opt* identifies options that were associated with this data unit, and *iov[0].iov_base* through *iov[iovcount-1].iov_base* contains the user data that was received. The return value of *t_rcvvudata*( ) is the number of bytes of user data given to the user.

**Note:** The limit on the total number of bytes available in all buffers passed (that is, *iov(0).iov_len* + . . + *iov(iovcount-1).iov_len*) may be constrained by implementation limits. If no other constraint applies, it will be limited by [INT_MAX]. In practice, the

availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter/gather functions.

By default, *t_rcvvudata*( ) operates in synchronous mode and waits for a data unit to arrive if none is currently available. However, if O_NONBLOCK is set (via *t_open*( ) or *fcntl*( )), *t_rcvvudata*( ) executes in asynchronous mode and fails if no data units are available.

If the buffers defined in the *iov[]* array are not large enough to hold the current data unit, the buffers will be filled and T_MORE will be set in flags on return to indicate that another *t_rcvvudata*( ) should be called to retrieve the rest of the data unit. Subsequent calls to *t_rcvvudata*( ) will return zero for the length of the address and options, until the full data unit has been received.

**VALID STATES**

T_IDLE

**ERRORS**

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

| | |
|---|---|
| [TBADDATA] | *iovcount* is greater than T_IOV_MAX. |
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TBUFOVFLW] | The number of bytes allocated for the incoming protocol address or options (*maxlen*) is greater than 0 but not sufficient to store the information. The unit data information to be returned in *unitdata* will be discarded. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNODATA] | O_NONBLOCK was set, but no data units are currently available from the transport provider. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUES**

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

**SEE ALSO**

*fcntl*( ), *t_alloc*( ), *t_open*( ), *t_rcvudata*( ), *t_rcvuderr*( ), *t_sndudata*( ), *t_sndvudata*( ).

**NAME**

t_snd - send data or expedited data over a connection

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *buf* | x (x) | = |
| *nbytes* | x | / |
| *flags* | x | / |

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

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

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

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

The sending of a zero-length fragment of a TSDU or ETSDU is only permitted where this is used to indicate the end of a TSDU or ETSDU; that is, when the T_MORE flag is not set. Some transport providers also forbid zero-length TSDUs and ETSDUs. See Appendix A on page 265 for a fuller explanation.

T_PUSH — If set in *flags*, requests that the provider transmit all data that it has accumulated but not sent. The request is a local action on the provider and does not affect any similarly named protocol flag (for example, the TCP PUSH flag). This effect of setting this flag is protocol-dependent, and it may be ignored entirely by transport providers which do not support the use of this feature.

**Note:** The communications provider is free to collect data in a send buffer until it accumulates a sufficient amount for transmission.

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

On successful completion, *t_snd*( ) returns the number of bytes (octets) accepted by the communications provider. Normally this will equal the number of octets specified in nbytes. However, if O_NONBLOCK is set or the function is interrupted by a signal, it is possible that only part of the data has actually been accepted by the communications provider. In this case, *t_snd*( ) returns a value that is less than the value of nbytes. If *t_snd*( ) is interrupted by a signal before it could transfer data to the communications provider, it returns –1 with *t_errno* set to [TSYSERR] and *errno* set to [EINTR].

If nbytes is zero and sending of zero bytes is not supported by the underlying communications service, *t_snd*( ) returns –1 with *t_errno* set to [TBADDATA].

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

The error [TLOOK] is returned for asynchronous events. It is required only for an incoming disconnect event but may be returned for other events.

**VALID STATES**

T_DATAXFER, T_INREL

**ERRORS**

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

[TBADDATA] Illegal amount of data:

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

— A send of a zero byte TSDU (ETSDU) or zero byte fragment of a TSDU (ETSDU) is not supported by the provider (see Appendix A on page 265.

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

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

| | |
|---|---|
| [TBADFLAG] | An invalid flag was specified. |
| [TFLOW] | O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**Note:** If the number of bytes accepted by the communications provider is less than the number of bytes requested, this may either indicate that O_NONBLOCK is set and the communications provider is blocked due to flow control, or that O_NONBLOCK is clear and the function was interrupted by a signal.

**SEE ALSO**

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

**CAVEATS**

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

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

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

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_snddis - send user-initiated disconnection request

**SYNOPSIS**

```
#include <xti.h>

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

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *call→addr.maxlen* | = | = |
| *call→addr.len* | = | = |
| *call→addr.buf* | = | = |
| *call→opt.maxlen* | = | = |
| *call→opt.len* | = | = |
| *call→opt.buf* | = | = |
| *call→udata.maxlen* | = | = |
| *call→udata.len* | x | = |
| *call→udata.buf* | ?(?) | = |
| *call→sequence* | ? | = |

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

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

The values in *call* have different semantics, depending on the context of the call to *t_snddis*( ). When rejecting a connection request, *call* must be non-null and contain a valid value of *sequence* to uniquely identify the rejected connection 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 disconnection request. The *addr*, *opt* and *sequence* fields of the **t_call** structure are ignored. If the user does not wish to send data to the remote user, the value of *call* may be a null pointer.

The *udata* structure specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider, as returned in the *discon* field, of the *info* argument of *t_open*( ) or *t_getinfo*( ). If the *len* field of *udata* is zero, no data will be sent to the remote user.

**VALID STATES**

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

**ERRORS**

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

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

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

| | |
|---|---|
| [TBADSEQ] | An invalid sequence number was specified, or a null *call* pointer was specified, when rejecting a connection request. |
| [TLOOK] | An asynchronous event, which requires attention, has occurred. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

*t_connect*( ), *t_getinfo*( ), *t_listen*( ), *t_open*( ).

**CAVEATS**

*t_snddis*( ) is an abortive disconnection. Therefore a *t_snddis*( ) issued on a connection endpoint may cause data previously sent via *t_snd*( ), or data not yet received, to be lost (even if an error is returned).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_sndrel - initiate an orderly release

**SYNOPSIS**

```
#include <xti.h>

int t_sndrel(int fd);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |

For transport providers of type T_COTS_ORD, this function is used to initiate an orderly release of the outgoing direction of data transfer and indicates to the transport provider that the transport user has no more data to send. The argument *fd* identifies the local transport endpoint where the connection exists. After calling *t_sndrel*( ), the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has not been received. For transport providers of types other than T_COTS_ORD, this function fails with error [TNOTSUPPORT].

**VALID STATES**

T_DATAXFER,T_INREL

**ERRORS**

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

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

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

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

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

[TOUTSTATE] The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

[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 error *(t_errno)*.

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

**RETURN VALUE**

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

**SEE ALSO**

*t_getinfo*( ), *t_open*( ), *t_rcvrel*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_sndreldata - initiate/respond to an orderly release with user data

**SYNOPSIS**

```
#include <xti.h>

int t_sndreldata(int fd, struct t_discon *discon);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *discon→udata.maxlen* | / | = |
| *discon→udata.len* | x | / |
| *discon→udata.buf* | ?(?) | / |
| *discon→reason* | ? | / |
| *discon→sequence* | / | / |

This function is used to initiate an orderly release of the outgoing direction of data transfer and to send user data with the release. The argument *fd* identifies the local transport endpoint where the connection exists, and *discon* points to a **t_discon** structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

After calling *t_sndreldata*( ), the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has not been received.

The field *reason* specifies the reason for the disconnection through a protocol-dependent *reason code*, and *udata* identifies any user data that is sent with the disconnection; the field *sequence* is not used.

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 or if the provider did not return T_ORDRELDATA in the *t_open*( ) flags, no data will be sent to the remote user.

If a user does not wish to send data and reason code to the remote user, the value of *discon* may be a null pointer.

This function is an optional service of the transport provider, only supported by providers of service type T_COTS_ORD. The flag T_ORDRELDATA in the *info→flag* field returned by *t_open*( ) or *t_getinfo*( ) indicates that the provider supports orderly release user data.

This function may not be available on all systems.

**VALID STATES**

T_DATAXFER, T_INREL

**ERRORS**

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

[TBADDATA] The amount of user data specified was not within the bounds allowed by the transport provider, or user data was supplied and the provider did not return T_ORDRELDATA in the *t_open*( ) flags.

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TFLOW] | O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting the function at this time. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNOTSUPPORT] | Orderly release is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

*t_getinfo*( ), *t_open*( ), *t_rcvreldata*( ), *t_rcvrel*( ), *t_sndrel*( ).

**NAME**

t_sndudata - send a data unit

**SYNOPSIS**

```
#include <xti.h>

int t_sndudata(int fd, const struct t_unitdata *unitdata);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *unitdata→addr.maxlen* | = | = |
| *unitdata→addr.len* | x | = |
| *unitdata→addr.buf* | x(x) | = |
| *unitdata→opt.maxlen* | = | = |
| *unitdata→opt.len* | x | = |
| *unitdata→opt.buf* | ?(?) | = |
| *unitdata→udata.maxlen* | = | = |
| *unitdata→udata.len* | x | = |
| *unitdata→udata.buf* | x(x) | = |

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

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

In *unitdata*, *addr* specifies the protocol address of the destination user, *opt* identifies options that the user wants associated with this request, and *udata* specifies the user data to be sent. The user may choose not to specify what protocol options are associated with the transfer by setting the *len* field of *opt* to zero. In this case, the provider uses the option values currently set for the communications endpoint.

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

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

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

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

If the call is interrupted, *t_sndudata*( ) will return [EINTR] and the datagram will not be sent.

**VALID STATES**

T_IDLE

**ERRORS**

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

| | |
|---|---|
| [TBADADDR] | The specified protocol address was in an incorrect format or contained illegal information. |
| [TBADDATA] | Illegal amount of data. A single send was attempted specifying a TSDU greater than that specified in the *info* argument, or a send of a zero byte TSDU is not supported by the provider. |
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TBADOPT] | The specified options were in an incorrect format or contained illegal information. |
| [TFLOW] | O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TOUTSTATE] | The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid. |
| [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 error *(t_errno)*. |
| [TSYSERR] | A system error has occurred during execution of this function. |

**RETURN VALUE**

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

**SEE ALSO**

*fcntl*( ), *t_alloc*( ), *t_open*( ), *t_rcvudata*( ), *t_rcvuderr*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_sndv — send data or expedited data, from one or more non-contiguous buffers, on a connection

**SYNOPSIS**

```
#include <xti.h>

int t_sndv(int fd, const struct t_iovec *iov,
    unsigned int iovcount, int flags);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *iovec* | x | / |
| *iovcount* | x | / |
| *iov[0].iov_base* | x(x) | =(=) |
| *iov[0].iov_len* | x | = |
| . . . . | | |
| *iov[iovcount-1].iov_base* | x(x) | =(=) |
| *iov[iovcount-1].iov_len* | x | = |
| *flags* | x | / |

This function is used to send either normal or expedited data. The argument *fd* identifies the local transport endpoint over which data should be sent, *iov* points to an array of buffer address/buffer length pairs. *t_sndv*( ) sends data contained in buffers *iov[0]*, *iov[1]*, through *iov[iovcount-1]*. *iovcount* contains the number of non-contiguous data buffers which is greater than zero and limited to T_IOV_MAX (an implementation-defined value of at least 16). If the limit is exceeded, or if the value is zero, the function fails with [TBADDATA].

**Note:** The limit on the total number of bytes available in all buffers passed (that is: *iov(0).iov_len* + . . + *iov(iovcount-1).iov_len*) may be constrained by implementation limits. If no other constraint applies, it will be limited by [INT_MAX]. In practice, the availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter/gather functions.

The argument *flags* specifies any optional flags described below:

T_EXPEDITED

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

T_MORE

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

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

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

T_PUSH

If set in *flags*, requests that the provider transmit all data that it has accumulated but not sent. The request is a local action on the provider and does not affect any similarly named protocol flag (for example, the TCP PUSH flag). This effect of setting this flag is protocol-dependent, and it may be ignored entirely by transport providers which do not support the use of this feature.

**Note:** The communications provider is free to collect data in a send buffer until it accumulates a sufficient amount for transmission.

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

On successful completion, *t_sndv*( ) returns the number of bytes accepted by the transport provider. Normally this will equal the total number of bytes to be sent, that is,

*(iov[0].iov_len + . . + iov[iovcount-1].iov_len)*

However, the interface is constrained to send at most INT_MAX bytes in a single send. When *t_sndv*( ) has submitted INT_MAX (or lower constrained value, see the note above) bytes to the provider for a single call, this value is returned to the user. However, if O_NONBLOCK is set or the function is interrupted by a signal, it is possible that only part of the data has actually been accepted by the communications provider. In this case, *t_sndv*( ) returns a value that is less than the value of nbytes. If *t_sndv*( ) is interrupted by a signal before it could transfer data to the communications provider, it returns –1 with *t_errno* set to [TSYSERR] and *errno* set to [EINTR].

If the number of bytes of data in the *iov* array is zero and sending of zero octets is not supported by the underlying transport service, *t_sndv*( ) returns –1 with *t_errno* set to [TBADDATA].

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

The error [TLOOK] is returned for asynchronous events. It is required only for an incoming disconnect event but may be returned for other events.

**VALID STATES**

T_DATAXFER, T_INREL

**ERRORS**

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

[TBADDATA] Illegal amount of data:

- A single send was attempted specifying a TSDU (ETSDU) or fragment TSDU (ETSDU) greater than that specified by the current values of the TSDU or ETSDU fields in the *info* argument.
- A send of a zero byte TSDU (ETSDU) or zero byte fragment of a TSDU (ETSDU) is not supported by the provider.

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

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

[TBADFLAG] An invalid flag was specified.

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

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

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

[TOUTSTATE] The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

[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 error *(t_errno)*.

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

**RETURN VALUES**

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

**Notes:**

1. In synchronous mode, if more than INT_MAX bytes of data are passed in the *iov* array, only the first INT_MAX bytes will be passed to the provider.
2. If the number of bytes accepted by the communications provider is less than the number of bytes requested, this may either indicate that O_NONBLOCK is set and the communications provider is blocked due to flow control, or that O_NONBLOCK is clear and the function was interrupted by a signal.

**SEE ALSO**

*t_getinfo*( ), *t_open*( ), *t_rcvv*( ), *t_rcv*( ), *t_snd*( ).

**CAVEATS**

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

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

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

**NAME**

t_sndvudata — send a data unit from one or more noncontiguous buffers

**SYNOPSIS**

```
#include <xti.h>

int t_sndvudata(int fd, struct t_unitdata *unitdata,
    struct t_iovec *iov, unsigned int iovcount);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *unitdata->addr.maxlen* | / | = |
| *unitdata->addr.len* | x | = |
| *unitdata->addr.buf* | x(x) | =(=) |
| *unitdata->opt.maxlen* | / | = |
| *unitdata->opt.len* | x | = |
| *unitdata->opt.buf* | ?(?) | =(=) |
| *unitdata->udata.maxlen* | / | = |
| *unitdata->udata.len* | / | = |
| *unitdata->udata.buf* | / | = |
| *iov[0].iov_base* | x(x) | =(=) |
| *iov[0].iov_len* | x | = |
| . . . . | | |
| *iov[iovcount-1].iov_base* | x(x) | =(=) |
| *iov[iovcount-1].iov_len* | x | = |
| *iovcount* | x | / |

This function is used in connectionless mode to send a data unit to another transport user. The argument *fd* identifies the local transport endpoint through which data will be sent, *iovcount* contains the number of non-contiguous udata buffers and is limited to an implementation-defined value given by T_IOV_MAX which is at least 16, and *unitdata* points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

If the limit on *iovcount* is exceeded or *iovcount* is zero, the function fails with [TBADDATA].

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

The data to be sent is identified by *iov[0]* through *iov[iovcount-1]*.

**Note:** The limit on the total number of bytes available in all buffers passed (that is: *iov(0).iov_len + . . + iov(iovcount-1).iov_len )* may be constrained by implementation limits. If no other constraint applies, it will be limited by [INT_MAX]. In practice, the availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter/gather functions.

By default, *t_sndvudata*( ) operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NONBLOCK is set (via *t_open*( ) or *fcntl*( )), *t_sndvudata*( ) executes

in asynchronous mode and will fail under such conditions. The process can arrange to be notified of the clearance of a flow control restriction via either *t_look*( ) or the EM interface.

If the amount of data specified in *iov[0]* through *iov[iovcount-1]* exceeds the TSDU size as returned in the *tsdu* field of the *info* argument of *t_open*( ) or *t_getinfo*( ), or is zero and sending of zero octets is not supported by the underlying transport service, a [TBADDATA] error is generated. If *t_sndvudata*( ) is called before the destination user has activated its transport endpoint (see *t_bind*( )), the data unit may be discarded.

If it is not possible for the transport provider to immediately detect the conditions that cause the errors [TBADDADDR] and [TBADOPT], these errors will alternatively be returned by *t_rcvuderr*( ). An application must therefore be prepared to receive these errors in both of these ways.

**VALID STATES**

T_IDLE

**ERRORS**

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

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

[TBADDATA] Illegal amount of data.

- A single send was attempted specifying a TSDU greater than that specified in the *info* argument, or a send of a zero byte TSDU is not supported by the provider.
- *iovcount* is zero or greater than T_IOV_MAX.

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

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

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

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

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

[TOUTSTATE] The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

[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 error *(t_errno)*.

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

**RETURN VALUES**

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

**SEE ALSO**

*fcntl*( ), *t_alloc*( ), *t_open*( ), *t_rcvudata*( ), *t_rcvvudata*( ), *t_rcvuderr*( ), *t_sndudata*( ).

**NAME**

t_strerror - produce an error message string

**SYNOPSIS**

```
#include <xti.h>

const char *t_strerror(int errnum);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *errnum* | x | / |

The *t_strerror*( ) function maps the error number in *errnum* that corresponds to an XTI error to a language-dependent error message string and returns a pointer to the string. The string pointed to will not be modified by the program, but may be overwritten by a subsequent call to the *t_strerror* function. The string is not terminated by a newline character. The language for error message strings written by *t_strerror*( ) is that of the current locale. If it is English, the error message string describing the value in *t_errno* may be derived from the comments following the *t_errno* codes defined in **<xti.h>**. If an error code is unknown, and the language is English, *t_strerror*( ) returns the string:

```
"<error>: error unknown"
```

where <error> is the error number supplied as input. In other languages, an equivalent text is provided.

**VALID STATES**

ALL - apart from T_UNINIT

**RETURN VALUE**

The function *t_strerror*( ) returns a pointer to the generated message string.

**SEE ALSO**

*t_error*( )

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_sync - synchronise transport library

**SYNOPSIS**

```
#include <xti.h>

int t_sync(int fd);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |

For the transport endpoint specified by *fd*, *t_sync*( ) synchronises the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert an uninitialised file descriptor (obtained via *open*( ), *dup*( ) or as a result of a *fork*( ) and *exec*( )) to an initialised transport endpoint, assuming that the file descriptor referenced a transport endpoint, by updating and allocating the necessary library data structures. This function also allows two cooperating processes to synchronise their interaction with a transport provider.

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

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

If the transport endpoint is undergoing a state transition when *t_sync*( ) is called, the function will fail.

**VALID STATES**

ALL - apart from T_UNINIT

**ERRORS**

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

[TBADF] The specified file descriptor does not refer to a transport endpoint. This error may be returned when the *fd* has been previously closed or an erroneous number may have been passed to the call.

[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 error *(t_errno)*.

[TSTATECHNG] The transport endpoint is undergoing a state change.

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

**RETURN VALUE**

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

| | |
|---|---|
| T_UNBND | Unbound. |
| T_IDLE | Idle. |
| T_OUTCON | Outgoing connection pending. |
| T_INCON | Incoming connection pending. |
| T_DATAXFER | Data transfer. |
| T_OUTREL | Outgoing orderly release (waiting for an orderly release indication). |
| T_INREL | Incoming orderly release (waiting for an orderly release request). |

**SEE ALSO**

*dup*( ), *exec*( ), *fork*( ), *open*( ).

**CHANGE HISTORY**

**Issue 4**

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

**NAME**

t_sysconf — get configurable XTI variables

**SYNOPSIS**

```
#include <xti.h>

int t_sysconf(int name);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *name* | x | / |

The *t_sysconf*( ) function provides a method for the application to determine the current value of configurable and implementation-dependent XTI limits or options.

The *name* argument represents the XTI system variable to be queried. The following table lists the minimal set of XTI system variables from **<xti.h>** that can be returned by *t_sysconf*( ), and the symbolic constants, defined in **<xti.h>** that are the corresponding values used for *name*.

| Variable | Value of Name |
|---|---|
| T_IOV_MAX | _SC_T_IOV_MAX |

**VALID STATES**

All.

**ERRORS**

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

[TBADFLAG] *name* has an invalid value.

**RETURN VALUES**

If *name* is valid, *t_sysconf*( ) returns the value of the requested limit/option (which might be –1) and leaves *t_errno* unchanged. Otherwise, a value of –1 is returned and *t_errno* is set to indicate an error.

**SEE ALSO**

*sysconf*(3C), *t_rcvv*( ), *t_rcvvudata*( ), *t_sndv*( ), *t_sndvudata*( ).

**NAME**

t_unbind - disable a transport endpoint

**SYNOPSIS**

```
#include <xti.h>

int t_unbind(int fd);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |

The *t_unbind*( ) function disables the transport endpoint specified by *fd* which was previously bound by *t_bind*( ). On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider. An endpoint which is disabled by using *t_unbind*( ) can be enabled by a subsequent call to *t_bind*( ).

**VALID STATES**

T_IDLE

**ERRORS**

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

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

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

[TOUTSTATE] The communications endpoint referenced by *fd* is not in one of the states in which a call to this function is valid.

[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 error *(t_errno)*.

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

**RETURN VALUE**

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

**SEE ALSO**

*t_bind*( ).

**CHANGE HISTORY**

**Issue 4**

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

Chapter 15

# The <xti.h> Header

This chapter describes the effects of including the <xti.h> header file, which is made available by each XTI implementation and is included by applications that use the XTI functions. The data definitions and macros specified in this chapter need not be contained in **<xti.h>** itself, but must be available to the application when **<xti.h>** is included.

There is an example **<xti.h>** header file in Appendix E on page 317.

**Note:** Applications written to compilation environments earlier than those required by this issue of the specification (see Section 1.3 on page 3) and defining _XOPEN_SOURCE to be less than 500 may have the data structures and constants of certain protocol specific headers automatically exposed by the inclusion of **<xti.h>** for compatibility. The individual protocol-specific appendices document the providers for which this may be the case.

Certain symbols or macros may be exposed to applications including **<xti.h>** for compatibility with applications transitioning from older issues of this specification where their semantics are specified. Exposing these symbols or macros is allowed but not required. Symbols or macros that may be exposed in this implementation-dependent manner are:

**OPT_NEXTHDR**
**T_ALIGN**[13]
All protocol specific symbols exposed through **<xti_osi.h>** as specified in Appendix A on page 265.
All protocol-specific symbols exposed through **<xti_inet.h>** as specified in Chapter 16 on page 251.

The function definitions in this chapter conform to the ISO C standard (see referenced documents).

13. T_ALIGN macro was used only in declaring OPT_NEXTHDR and not formally specified in XNS4, but XNS4 XTI header test assertions required it. It is removed as in spirit it is linked to not requiring OPT_NEXTHDR.

**NAME**

**SYNOPSIS**

```
#include <xti.h>
```

**DESCRIPTION**

**<xti.h>** makes available the definitions of **t_scalar_t** and **t_uscalar_t** respectively as a signed and unsigned opaque integral type of equal length of at least 32 bits.[14]

Quantity *t_errno* is defined as a modifiable lvalue of type int. See Section 10.6 on page 123.

The following symbols are defined for the errors reported by XTI.

| Symbol | Error |
|---|---|
| TBADADDR | incorrect address format |
| TBADOPT | incorrect option format |
| TACCES | incorrect permissions |
| TBADF | illegal fd |
| TNOADDR | could not allocate address |
| TOUTSTATE | out of state |
| TBADSEQ | bad call sequence number |
| TSYSERR | system error |
| TLOOK | event requires attention |
| TBADDATA | illegal amount of data |
| TBUFOVFLW | buffer not large enough |
| TFLOW | flow control |
| TNODATA | no data |
| TNODIS | disconnection indication not found on queue |
| TNOUDERR | unitdata error not found |
| TBADFLAG | bad flags |
| TNOREL | no orderly release event found on queue |
| TNOTSUPPORT | primitive/action not supported |
| TSTATECHNG | state is in process of changing |
| TNOSTRUCTYPE | unsupported structure type requested |
| TBADNAME | invalid transport provider name |
| TBADQLEN | qlen is zero |
| TADDRBUSY | address in use |
| TINDOUT | outstanding connection indications |
| TPROVMISMATCH | transport provider mismatch |
| TRESQLEN | resfd specified to t_accept() with qlen >0 |
| TRESADDR | resfd not bound to same addr as fd |
| TQFULL | incoming connection queue full |
| TPROTO | XTI protocol error |

14. To forestall portability problems, it is recommended that applications should not use values larger than 2**32 –1.

The following symbols are defined with bitwise distinct values for the XTI events described in Chapter 12 on page 141.

| Symbol | Event |
|---|---|
| T_LISTEN | connection indication received |
| T_CONNECT | connection confirmation received |
| T_DATA | normal data received |
| T_EXDATA | expedited data received |
| T_DISCONNECT | disconnection received |
| T_UDERR | datagram error indication |
| T_ORDREL | orderly release indication |
| T_GODATA | sending normal data is again possible |
| T_GOEXDATA | sending expedited data is again possible |

The following symbols are defined for the flags used by the XTI functions. The values of T_MORE, T_EXPEDITED and T_PUSH are bitwise distinct. The values of the other symbols are bitwise distinct from each other, but not necessarily bitwise distinct from the values of T_MORE, T_EXPEDITED and T_PUSH.

| Symbol | Value |
|---|---|
| T_MORE | more data |
| T_EXPEDITED | expedited data |
| T_PUSH | send data immediately |
| T_NEGOTIATE | set options |
| T_CHECK | check options |
| T_DEFAULT | get default options |
| T_SUCCESS | successful |
| T_FAILURE | failure |
| T_CURRENT | get current options |
| T_PARTSUCCESS | partial success |
| T_READONLY | read-only |
| T_NOTSUPPORT | not supported |

The XTI functions are defined as follows:

```
extern int t_accept(int, int, const struct t_call *);
extern void *t_alloc(int, int, int);
extern int t_bind(int, const struct t_bind *, struct t_bind *);
extern int t_close(int);
extern int t_connect(int, const struct t_call *, struct t_call *);
extern int t_error(const char *);
extern int t_free(void *, int);
extern int t_getinfo(int, struct t_info *);
extern int t_getprotaddr(int, struct t_bind *, struct t_bind *);
extern int t_getstate(int);
extern int t_listen(int, struct t_call *);
extern int t_look(int);
extern int t_open(const char *, int, struct t_info *);
extern int t_optmgmt(int, const struct t_optmgmt *, struct t_optmgmt *);
extern int t_rcv(int, void *, unsigned int, int *);
extern int t_rcvconnect(int, struct t_call *);
extern int t_rcvdis(int, struct t_discon *);
extern int t_rcvrel(int);
extern int t_rcvreldata(int, struct t_discon *);
extern int t_rcvudata(int, struct t_unitdata *, int *);
extern int t_rcvuderr(int, struct t_uderr *);
extern int t_rcvv(int, struct t_iovec *, unsigned int, int *);
extern int t_rcvvudata(int, struct t_unitdata *, struct t_iovec *,
                       unsigned int, int *);
extern int t_snd(int, const void *, unsigned int, int);
extern int t_snddis(int, const struct t_call *);
extern int t_sndrel(int);
extern int t_sndreldata(int, const struct t_discon *);
extern int t_sndudata(int, const struct t_unitdata *);
extern int t_sndv(int, const struct t_iovec *, unsigned int, int);
extern int t_sndvudata(int, const struct t_unitdata *,
                       const struct t_iovec *, unsigned int);
extern const char *t_strerror(int);
extern int t_sync(int);
extern int t_sysconf(const int);
extern int t_unbind(int);
```

Structure type **t_info** is defined for protocol-specific service limits. It has the following members.

| Member | Type | Contents |
|---|---|---|
| addr | t_scalar_t | maximum size of the transport protocol address |
| options | t_scalar_t | maximum number of bytes of protocol-specific options |
| tsdu | t_scalar_t | maximum size of a TSDU |
| etsdu | t_scalar_t | maximum size of ETSDU |
| connect | t_scalar_t | maximum amount of data allowed on connection establishment functions |
| discon | t_scalar_t | maximum data allowed on t_rcvdis, t_snddis, t_rcvreldata and t_sndreldata functions |
| servtype | t_scalar_t | service type supported by transport provider |
| flags | t_scalar_t | other information about the transport provider |

The following integer symbolic constants are defined with distinct values for service types:

| Symbol | Service |
|---|---|
| T_COTS | connection-mode transport service |
| T_COTS_ORD | connection-mode with orderly release |
| T_CLTS | connectionless-mode transport service |

The following symbols are defined with values that are bitwise-distinct bit masks for the flags field of a structure of type **struct**t_info**,** giving other information about the transport provider:

| Symbol | Information |
|---|---|
| T_ORDRELDATA | supports orderly release data |
| T_SENDZERO | supports zero length TSDUs |

Structure type **netbuf** is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| maxlen | unsigned int | maximum length of data in octets |
| len | unsigned int | length of data in octets |
| buf | void * | data |

Structure type **t_opthdr** is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| len | t_uscalar_t | total length of option: sizeof (struct t_opthdr) + length of value in bytes |
| level | t_uscalar_t | protocol affected |
| name | t_uscalar_t | option name |
| status | t_uscalar_t | status value |

Structure type **t_bind** is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| addr | struct netbuf | protocol address |
| qlen | unsigned int | maximum number of outstanding connection indications |

Structure type **t_optmgmt** is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| opt | struct netbuf | options |
| flags | t_scalar_t | actions |

Structure type **t_discon** is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| udata | struct netbuf | user data |
| reason | int | reason code |
| sequence | int | sequence number |

Structure type **t_call** is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| addr | struct netbuf | address |
| opt | struct netbuf | options |
| udata | struct netbuf | user data |
| sequence | int | sequence number |

Structure type **t_unitdata** is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| addr | struct netbuf | address |
| opt | struct netbuf | options |
| udata | struct netbuf | user data |

Structure type **t_uderr** is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| addr | struct netbuf | address |
| opt | struct netbuf | options |
| error | t_scalar_t | error code |

Structure type **t_iovec** is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| iov_base | void * | data |
| iov_len | size_t | length in bytes |

The following symbolic integer constant is defined with a value of at least 16 for the maximum number of buffers that can be passed to t_rcvv(),t_rcvvudata(),t_sndv()ort_sndvudata():

T_IOV_MAX

The following integer symbolic constants are defined with distinct values to indicate structure types when dynamically allocating structures via t_alloc ().

| Symbol | Structure Type |
|---|---|
| T_BIND | struct t_bind |
| T_OPTMGMT | struct t_optmgmt |
| T_CALL | struct t_call |
| T_DIS | struct t_discon |
| T_UNITDATA | struct t_unitdata |
| T_UDERROR | struct t_uderr |
| T_INFO | struct t_info |

The following symbols are defined with values that are bit masks that specify which fields of the above structures should be allocated by *t_alloc*( ). The value of T_ALL is the bitmask with all bits set. The other values are bitwise distinct.

| Symbol | Value | Field |
|---|---|---|
| T_ADDR | 0x01 | address |
| T_OPT | 0x02 | options |
| T_UDATA | 0x04 | user data |
| T_ALL | 0xffff | all the above fields supported |

The following symbolic integer constants are defined with distinct values, representing the states described in Chapter 12 on page 141:

| Symbol | State |
|---|---|
| T_UNBND | unbound |
| T_IDLE | idle |
| T_OUTCON | outgoing connection pending |
| T_INCON | incoming connection pending |
| T_DATAXFER | data transfer |
| T_OUTREL | outgoing release pending |
| T_INREL | incoming release pending |

The following symbolic integer constants are defined. The values of T_YES and T_NO are distinct. The values of T_INFINITE and T_INVALID are distinct and are both less than zero. Their definitions are protected by parentheses to ensure that they are interpreted correctly when the symbols are used in expressions:

T_NO
T_NULL
T_ABSREQ
T_INFINITE
T_INVALID

The following symbolic integer constant is defined for the name of system variable T_IOV_MAX:

_SC_T_IOV_MAX

The following symbolic integer constants are defined:

T_UNSPEC
T_ALLOPT

The values of T_UNSPEC, T_ALLOPT and the values of all constants defined to identify protocol levels and options are all different. The value of T_UNSPEC is applicable to any integer type.

Macros T_OPT_FIRSTHDR(nbp), T_OPT_NEXTHDR(nbp, tohp), and T_OPT_DATA(tohp), as specified on the man page for *t_optmgmt*( ), are defined.

Integer symbolic constant XTI_GENERIC is defined for the protocol level of XTI.

The following **t_uscalar_t** integer symbolic constants are defined with distinct values for XTI-level options:

| **Symbol** | **Meaning** |
|---|---|
| XTI_DEBUG | enable debugging |
| XTI_LINGER | linger on close if data present |
| XTI_RCVBUF | receive buffer size |
| XTI_RCVLOWAT | receive low-water mark |
| XTI_SNDBUF | send buffer size |
| XTI_SNDLOWAT | send low-water mark |

Structure type **t_linger** is defined with the following members:

| **Member** | **Type** | **Contents** |
|---|---|---|
| l_onoff | t_scalar_t | option on/off |
| l_linger | t_scalar_t | linger time |

**SEE ALSO**

*t_accept*( ), *t_alloc*( ), *t_bind*( ), *t_close*( ), *t_connect*( ), *t_error*( ), *t_free*( ), *t_getinfo*( ), *t_getprotaddr*( ), *t_getstate*( ), *t_listen*( ), *t_look*( ), *t_open*( ), *t_optmgmt*( ), *t_rcv*( ), *t_rcvconnect*( ), *t_rcvdis*( ), *t_rcvrel*( ), *t_rcvreldata*( ), *t_rcvudata*( ), *t_rcvuderr*( ), *t_rcvv*( ), *t_rcvvudata*( ), *t_snd*( ), *t_snddis*( ), *t_sndrel*( ), *t_sndreldata*( ), *t_sndudata*( ), *t_sndv*( ), *t_sndvudata*( ), *t_strerror*( ), *t_sync*( ), *t_sysconf*( ), *t_unbind*( ), **<xti_inet.h>**, **<xti_osi.h>**.

**CHANGE HISTORY**

First released in Issue 5.2.

Chapter 16

# Use of XTI with Internet Protocols

## 16.1 Introduction

This Chapter describes the protocol-specific information that is relevant for TCP and UDP transport providers. It also defines data structures and constants required for TCP and UDP transport providers which are exposed through the **<xti_inet.h>** header file.

**Note:** Applications written to compilation environments earlier than those required by this issue of the specification (see Section 1.3 on page 3) and defining _XOPEN_SOURCE to be less than 500, may have these data structures and constants exposed through the inclusion of **<xti.h>**.

## 16.2 Protocol Features

### T_MORE Flag and TSDUs

The notion of TSDU is not supported by TCP transport provider, so the T_MORE flag will be ignored when TCP is used.

### Expedited Data

TCP does not have a notion of expedited data in a sense comparable to ISO expedited data. TCP defines an urgent mechanism, by which in-line data is marked for urgent delivery. T_UDP has no urgent mechanism. See the TCP Standard for more detailed information.

### Orderly Release

The orderly release functions *t_sndrel*( ) and *t_rcvrel*( ) were defined to support the orderly release facility of TCP. They are the recommended means of releasing a TCP connection. The specification of TCP states that only established connections may be closed with orderly release: that is, on an endpoint in T_DATAXFER or T_INREL state.

### Abortive Release

Functions *t_snddis*( ) and *t_rcvdis*( ) may be used to perform abortive release over TCP transport. However their use is not recommended as the abortive release primitive (RST segment) is not transmitted reliably by the TCP protocol.

### Connection Establishment

TCP does not allow the possibility of refusing a connection indication. Each connection indication causes the TCP transport provider to establish the connection. Therefore, *t_listen*( ) and *t_accept*( ) have a semantic which is slightly different from that for ISO providers.

### Connection Release

After a connection has been released, the local address bound to the endpoint may be qualified with the local IP address (rather than having a wildcard IP address). Also, the port number itself may have been changed during the *t_accept*( ) processing. If the endpoint is not being reused immediately then it is recommended that it should be unbound or closed so that other users can successfully bind to the address.

## 16.3 Options

Options are formatted according to the structure **t_opthdr** as described in Chapter 13. A transport provider compliant to this specification supports none, all or any subset of the options defined in Section 16.3.1, Section 16.3.2 and Section 16.3.3. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode.

### 16.3.1 TCP-level Options

The protocol level is T_INET_TCP. For this level, Table 16-1 shows the options that are defined.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_TCP_KEEPALIVE | `struct t_kpalive` | see text | check if connections are alive |
| T_TCP_MAXSEG | `t_uscalar_t` | length in octets | get TCP maximum segment size |
| T_TCP_NODELAY | `t_uscalar_t` | T_YES/T_NO | don't delay send to coalesce packets |

**Table 16-1** TCP-level Options

These options do *not* have end-to-end significance. They may be negotiated in all XTI states except T_UNBND and T_UNINIT. They are read-only in state T_UNBND. See Chapter 13 for the difference between options that have end-to-end significance and those that do not.

#### Absolute Requirements

A request for T_TCP_NODELAY and a request to activate T_TCP_KEEPALIVE is an absolute requirement. T_TCP_MAXSEG is a read-only option.

#### Further Remarks

T_TCP_KEEPALIVE If this option is set, a keep-alive timer is activated to monitor idle connections that might no longer exist. If a connection has been idle since the last keep-alive timeout, a keep-alive packet is sent to check if the connection is still alive or broken.

Keep-alive packets are not an explicit feature of TCP, and this practice is not universally accepted. According to RFC 1122 (see Referenced Documents):

''a keep-alive mechanism should only be invoked in server applications that might otherwise hang indefinitely and consume resources unnecessarily if a client crashes or aborts a connection during a network failure''.

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

```
struct t_kpalive {
    t_scalar_t kp_onoff;    /* switch option on/off */
    t_scalar_t kp_timeout;  /* keep-alive timeout   */
                            /* in minutes           */
}
```

Legal values for the field *kp_onoff* are:

T_NO switch keep-alive timer off
T_YES activate keep-alive timer

The field *kp_timeout* 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 the referenced RFC 1122). Legal values for this field are T_UNSPEC and all positive numbers.

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.

The use of this option might be restricted to privileged users.

T_TCP_MAXSEG This option is read-only. It is used to retrieve the maximum TCP segment size.

T_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. T_TCP_NODELAY is used to defeat this algorithm. Legal option values are T_YES (''don't delay'') and T_NO (''delay'').

### 16.3.2 T_UDP-level Options

The protocol level is T_INET_UDP. The option defined for this level is shown in Table 16-2.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_UDP_CHECKSUM | `t_uscalar_t` | T_YES/T_NO | checksum computation |

**Table 16-2** T_UDP-level Option

This option has end-to-end significance. It may be negotiated in all XTI states except T_UNBND and T_UNINIT. It is read-only in state T_UNBND. See Chapter 13 for the difference between options that have end-to-end significance and those that do not.

#### Absolute Requirements

A request for this option is an absolute requirement.

#### Further Remarks

T_UDP_CHECKSUM The option allows disabling/enabling of the T_UDP checksum computation. The legal values are T_YES (checksum enabled) and T_NO (checksum disabled).

If this option is returned with *t_rcvudata*( ), its value indicates whether a checksum was present in the received datagram or not.

Numerous cases of undetected errors have been reported when applications chose to turn off checksums for efficiency. The advisability of ever turning off the checksum check is very controversial.

### 16.3.3 T_IP-level Options

The protocol level is T_INET_IP. The options defined for this level are listed in Table 16-3.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_IP_BROADCAST | unsigned int | T_YES/T_NO | permit sending of broadcast messages |
| T_IP_DONTROUTE | unsigned int | T_YES/T_NO | just use interface addresses |
| T_IP_OPTIONS | array of unsigned characters | see text | T_IP per-packet options |
| T_IP_REUSEADDR | unsigned int | T_YES/T_NO | allow local address reuse |
| T_IP_TOS | unsigned char | see text | T_IP per-packet type of service |
| T_IP_TTL | unsigned char | time in seconds | T_IP per packet time-to-live |

**Table 16-3** T_IP-level Options

T_IP_OPTIONS and T_IP_TOS are both options with end-to-end significance. All other options do *not* have end-to-end significance. See Chapter 13 for the difference between options with end-to-end significance and options without.

T_IP_REUSEADDR may be negotiated in all XTI states except T_UNINIT. All other options may be negotiated in all other XTI states except T_UNBND and T_UNINIT; they are read-only in the state T_UNBND.

#### Absolute Requirements

A request for any of these options is an absolute requirement.

#### Further Remarks

T_IP_BROADCAST This option requests permission to send broadcast datagrams. It was defined to make sure that broadcasts are not generated by mistake. The use of this option is often restricted to privileged users.

T_IP_DONTROUTE This option indicates that outgoing messages should bypass the standard routing facilities. It is mainly used for testing and development.

T_IP_OPTIONS This option is used to set (retrieve) the OPTIONS field of each outgoing (incoming) IP datagram. Its value is a string of octets composed of a number of T_IP options, whose format matches those defined in the IP specification with one exception: the list of addresses for the source routing options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use.

The option is disabled if it is specified with ‘‘no value’’; that is, with an option header only.

The functions *t_connect*( ) (in synchronous mode), *t_listen*( ), *t_rcvconnect*( ) and *t_rcvudata*( ) return the OPTIONS field, if any, of the received IP datagram associated with this call. The function *t_rcvuderr*( ) returns the OPTIONS field of the data unit previously sent that produced the error. The function *t_optmgmt*( ) with T_CURRENT set retrieves the currently effective T_IP_OPTIONS that is sent with outgoing datagrams.

Common applications never need this option. It is mainly used for network debugging and control purposes.

T_IP_REUSEADDR Many TCP implementations do not allow the user to bind more than one transport endpoint to addresses with identical port numbers. If T_IP_REUSEADDR is set to T_YES this restriction is relaxed in the sense that it is now allowed to bind a transport endpoint to an address with a port number and an underspecified internet address (''wild card'' address) and further endpoints to addresses with the same port number and (mutually exclusive) fully specified internet addresses.

T_IP_TOS This option is used to set (retrieve) the *type-of-service* field of an outgoing (incoming) T_IP datagram. This field can be constructed by any OR'ed combination of one of the precedence flags and the type-of-service flags T_LDELAY, T_HITHRPT and T_HIREL:

— Precedence:

These flags specify datagram precedence, allowing senders to indicate the importance of each datagram. They are intended for Department of Defense applications. Legal flags are:

T_ROUTINE
T_PRIORITY
T_IMMEDIATE
T_FLASH
T_OVERRIDEFLASH
T_CRITIC_ECP
T_INETCONTROL
T_NETCONTROL.

Applications using T_IP_TOS but not the precedence level should use the value T_ROUTINE for precedence.

— Type of service:

These flags specify the type of service the IP datagram desires. Legal flags are:

| | |
|---|---|
| T_NOTOS | requests no distinguished type of service |
| T_LDELAY | requests low delay |
| T_HITHRPT | requests high throughput |
| T_HIREL | requests high reliability |

The option value is set using the macro SET_TOS(*prec,tos*), where *prec* is set to one of the precedence flags and *tos* to one or an OR'ed combination of the type-of-service flags. SET_TOS( ) returns the option value.

The functions *t_connect*( ), *t_listen*( ), *t_rcvconnect*( ) and *t_rcvudata*( ) return the *type-of-service* field of the received IP datagram associated with this call. The function *t_rcvuderr*( ) returns the *type-of-service* field of the data unit previously sent that produced the error.

The function *t_optmgmt*( ) with T_CURRENT set retrieves the currently effective T_IP_TOS value that is sent with outgoing datagrams.

The requested *type-of-service* cannot be guaranteed. It is a hint to the routing algorithm that helps it choose among various paths to a destination. Note also, that most hosts and gateways in the Internet these days ignore the *type-of-service* field.

T_IP_TTL — This option[15] is used to set the *time-to-live* field in an outgoing IP datagram. It specifies how long, in seconds, the datagram is allowed to remain in the Internet. The *time-to-live* field of an incoming datagram is not returned by any function (since it is not an option with end-to-end significance).

15. This is a simplified description. Refer to RFC 1122 (see Referenced Documents) for precise details.

## 16.4 Functions

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

Since user data cannot be exchanged during the connection establishment phase, *call->udata.len* must be set to 0.

When (*resfd != fd*), the function *t_accept*( ) is recommended to be called with *resfd* in T_UNBND state, though an endpoint which is bound to any local address (in T_IDLE state) can also be used.

After *t_accept*( ) completes, the endpoint resfd will represent a connected TCP endpoint whose complete binding essentially has both local and remote address components.

If file descriptor *resfd* was unbound before calling *t_accept*( ), after the call completes its local address binding would be to the same protocol address bound to *fd*. If file descriptor *resfd* was bound to a local address before calling *t_accept*( ), that local address binding is dissolved and the local address part of the binding after *t_accept*( ) completes would become same as the address bound to *fd*.

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

*t_bind*( ) The *addr* field of the **t_bind** structure represents the local socket; that is, an address which specifically includes a port identifier.

Some implementations treat port number 0 as a request to bind to any unused port. Other than that value, a port number part of the binding is specific. The IP address part of the binding can represent a single IP address or a wildcard binding to an address that could represent multiple IP addresses that are legal for the host.

*t_close*( ) The *t_close*( ) call will result in a *close* call on the descriptor of this XTI communication endpoint. If there are no other descriptors in this process or any other process which reference this communication endpoint, the *close*( ) call will perform an orderly connection termination according to the rules of a TCP CLOSE call on this connection endpoint as specified in standards RFC 793 and RFC 1122. If the XTI_LINGER option is supported and is used to enable the *linger* option, the linger time will affect the time an implementation lingers in the execution of *t_close*( ) or **close( )** A linger time of 0 specified with the XTI_LINGER option may cause an abortive release of a TCP connection, resulting in lost data.

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

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

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

Since user data cannot be exchanged during the connection establishment phase, *call->udata.maxlen* must be set to 0 before the call to *t_listen*( ). The *call->addr* structure contains the remote calling socket.

*t_look*( ) 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.

*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 transport protocol by setting fields in the **t_info** structure.

The following should be the values returned by the call to *t_open*( ) and *t_getinfo*( ) with the indicated transport providers.

| **Parameters** | **Before call** | **After call** | |
|---|---|---|---|
| | | T_TCP/T_IP | T_UDP/T_IP |
| *name* | x | / | / |
| *oflag* | x | / | / |
| *info->addr* | / | x | x |
| *info->options* | / | x | x |
| *info->tsdu* | / | 0 | x |
| *info->etsdu* | / | T_INFINITE | T_INVALID |
| *info->connect* | / | T_INVALID | T_INVALID |
| *info->discon* | / | T_INVALID | T_INVALID |
| *info->servtype* | / | T_COTS_ORD | T_CLTS |
| *info->flags* | / | 0 | T_SNDZERO |

'x' equals T_INVALID (–2) or an integral number greater than zero.

*t_rcv*( ) 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.

*t_rcvconnect*( ) 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 address of the remote calling endpoint.

*t_rcvdis*( ) Since data may not be sent with a disconnect, the *discon->udata* structure will not be meaningful.

*t_snd*( ) The T_MORE flag should be ignored. If *t_snd*( ) is called with more than one byte specified and with the T_EXPEDITED flag set, then the last byte of the buffer will be the byte pointed to by the TCP urgent pointer. If the T_EXPEDITED flag is set, at least one byte must be sent.

*Implementor's Note: Data for a t_snd( ) call with the T_EXPEDITED flag set may not pass data sent previously.*

*t_snddis*( ) Since data may not be sent with a disconnect, *call->udata.len* must be set to zero.

*t_sndudata*( ) Be aware that the maximum size of a connectionless-mode TSDU varies among implementations.

## 16.5 The <xti_inet.h> Header File

This section describes the effects of including the **<xti_inet.h>** header file, which is made available by each XTI implementation and is included by applications that use the XTI functions for communication via the Internet protocol suite.

The data definitions and macros specified in this section need not be contained in **<xti_inet.h>** itself, but must be available to the application when **<xti.h>** is included and **<xti_inet.h>** is included after **<xti.h>**.

**Note:** Applications written to compilation environments earlier than those required by this issue of the specification (see Section 1.3 on page 3) and defining _XOPEN_SOURCE to be less than 500 may have the data structures and constants of this header automatically exposed by the inclusion of **<xti.h>** for compatibility.

Certain symbols may be exposed to applications including **<xti_inet.h>** for compatibility with applications transitioning from older issues of this specification where their semantics are specified. Exposing these symbols is allowed but not required. Symbols that may be exposed in this implementation-dependent manner are:

INET_TCP
TCP_NODELAY
TCP_MAXSEG
TCP_KEEPALIVE
T_GARBAGE
INET_UDP
INET_IP
IP_OPTIONS
IP_TOS
IP_TTL
IP_REUSEADDR
IP_DONTROUTE
IP_BROADCAST

There is an example **<xti_inet.h>** header file in Appendix E on page 317.

**NAME**

**SYNOPSIS**

```
#include <xti_inet.h>
```

**DESCRIPTION**

The following symbol is defined to identify the ISO Transport protocol level:

| Symbol | Protocol Level |
|---|---|
| T_INET_TP | TCP |

The following symbols are defined with distinct integer values to identify TCP-level options:

| Symbol | Option |
|---|---|
| T_TCP_NODELAY | do not delay packets to coalesce |
| T_TCP_MAXSEG | get maximum segment size |
| T_TCP_KEEPALIVE | check whether connections are alive |

Structure type **t_kpalive** is defined with the following members for use with TCP_KEEPALIVE option:

| Member | Type | Contents |
|---|---|---|
| kp_onoff | t_scalar_t | option on/off |
| kp_timeout | t_scalar_t | timeout in minutes |

The following symbol is defined to identify the UDP protocol level:

| Symbol | Protocol Level |
|---|---|
| T_INET_UDP | UDP |

The following symbol is defined with integer value to identify the UDP-level checksum option:

| Symbol | Option |
|---|---|
| T_UDP_CHECKSUM | checksum computation |

The following symbol is defined to identify the IP protocol level:

| Symbol | Protocol Level |
|---|---|
| T_INET_IP | IP |

The following symbols are defined with distinct integer values to identify IP-level options:

| Symbol | Option |
|---|---|
| T_IP_OPTIONS | IP per-packet options |
| T_IP_TOS | IP per-packet type of service |
| T_IP_TTL | IP per-packet time to live |
| T_IP_REUSEADDR | allow local address reuse |
| T_IP_DONTROUTE | just use interface addresses |
| T_IP_BROADCAST | permit sending of broadcast messages |

The following symbols are defined with distinct integer values to identify IP Types of Service:

| Symbol | Type of Service |
|---|---|
| T_NOTOS | normal |
| T_LDELAY | low delay |
| T_HITHRPT | high throughput |
| T_HIREL | high resilience |
| T_LOCOST | low cost |

The following symbols are defined with distinct integer values to identify IP Type of Service precedence levels:

| Symbol | IP TOS Precedence Level |
|---|---|
| T_ROUTINE | routine |
| T_PRIORITY | priority |
| T_IMMEDIATE | immediate |
| T_FLASH | flash |
| T_OVERRIDEFLASH | override flash |
| T_CRITIC_ECP | critical/ECP |
| T_INETCONTROL | internetwork control |
| T_NETCONTROL | network control |

Macro SET_TOS(prec, tos) is defined such that SET_TOS(prec, tos) is the value of the IP Type of Service field for Type of Service *tos* and precedence *prec*.

**SEE ALSO**

**<xti.h>**, **<xti_osi.h>**.

**CHANGE HISTORY**

First released in Issue 5.2.

*Technical Standard*

# Networking Services (XNS) Issue 5.2

# Part 4: Appendixes

*The Open Group*

*Appendix A*

# *Use of XTI with ISO Transport Protocols*

## A.1 General

This appendix describes the protocol-specific information that is relevant for ISO transport providers. This appendix also describes the protocol-specific information that is relevant when ISO transport services are provided over a TCP network[16].

In general, this Appendix describes the characteristics that the ISO and ISO-over-TCP transport providers have in common, with notes indicating where they differ.

**Notes:**

1. Protocol address:

   In an ISO environment, the protocol address is the transport address.

2. Sending data of zero octets:

   The transport service definition, both in connection-mode and in connectionless-mode, does not permit sending a TSDU of zero octets. So, in connectionless-mode, if the *len* parameter is set to zero, the *t_sndudata*( ) call will always return unsuccessfully with –1 and *t_errno* set to [TBADDATA]. In connection-mode, if the *nbytes* parameter is set to zero, the *t_snd*( ) call will return with –1 and *t_errno* set to [TBADDATA] if either the T_MORE flag is set, or the T_MORE flag is not set and the preceding *t_snd*( ) call completed a TSDU or ETSDU (that is, the call has requested sending a zero byte TSDU or ETSDU).

3. An ISO-over-TCP transport provider does not provide the connectionless-mode.

This Appendix also defines the data structures and constants required for ISO and ISO-over-TCP transport providers which are exposed through the **<xti_osi.h>** header file.

**Note:** Applications written to compilation environments earlier than those required by this issue of the specification (see Section 1.3 on page 3) and defining _XOPEN_SOURCE to be less than 500, may have these data structures and constants exposed through the inclusion of **<xti.h>**.

16. The mapping for ISO-over-TCP that is referred to here is that defined by RFC-1006: *ISO Transport Service on top of the TCP*, Version 3, May 1987, Marshall T Rose and Dwight E Cass, Network Working Group, Northrop Research & Technology Center. See also The Open Group publication: *Guide to IPS-OSI Coexistence and Migration* , Document Number G140; the relevant sections are 4.6.2 (Implementation of OSI Services over IPS) and 4.6.3 (Comments).

## A.2 Options

Options are formatted according to the structure **t_opthdr** as described in Chapter 13. A transport provider compliant to this specification supports none, all or any subset of the options defined in Section A.2.1 and Section A.2.2 on page 270. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode. An ISO-over-TCP provider supports a subset of the options defined in Section A.2.1.

### A.2.1 Connection-mode Service

The protocol level of all subsequent options is ISO_TP.

All options are association-related, that is, they are options with end-to-end significance (see Chapter 13). They may be negotiated in the XTI states T_IDLE and T_INCON, and are read-only in all other states except T_UNINIT.

#### *A.2.1.1 Options for Quality of Service and Expedited Data*

These options are all defined in the ISO 8072:1986 transport service definition (see the ISO Transport references). The definitions are not repeated here.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_TCO_THROUGHPUT | `struct thrpt` | octets per second | throughput |
| T_TCO_TRANSDEL | `struct transdel` | time in milliseconds | transit delay |
| T_TCO_RESERRORRATE | `struct rate` | OPT_RATIO | residual error rate |
| T_TCO_TRANSFFAILPROB | `struct rate` | OPT_RATIO | transfer failure probability |
| T_TCO_ESTFAILPROB | `struct rate` | OPT_RATIO | connection establ. failure probability |
| T_TCO_RELFAILPROB | `struct rate` | OPT_RATIO | connection release failure probability |
| T_TCO_ESTDELAY | `struct rate` | time in milliseconds | connection establ. delay |
| T_TCO_RELDELAY | `struct rate` | time in milliseconds | connection release delay |
| T_TCO_CONNRESIL | `struct rate` | OPT_RATIO | connection resilience |
| T_TCO_PROTECTION | `t_uscalar_t` | see text | protection |
| T_TCO_PRIORITY | `t_uscalar_t` | see text | priority |
| T_TCO_EXPD | `t_uscalar_t` | T_YES/T_NO | expedited data |

**Table A-1** Options for Quality of Service and Expedited Data

OPT_RATIO is defined as OPT_RATIO = $-\log_{10}$(ratio). The *ratio* is dependent on the parameter, but is always composed of a number of failures divided by a total number of samples. This may be, for example, the number of TSDUs transferred in error divided by the total number of TSDU transfers (T_TCO_RESERRORRATE).

**Absolute Requirements**

For the options in Table A-1 on page 266, the transport user can indicate whether the request is an absolute requirement or whether a degraded value is acceptable. For the QOS options based on **struct rate** an absolute requirement is specified via the field *minacceptvalue*, if that field is given a value different from T_UNSPEC. The value specified for T_TCO_PROTECTION is an absolute requirement if the T_ABSREQ flag is set. The values specified for T_TCO_EXPD and T_TCO_PRIORITY are never absolute requirements.

**Further Remarks**

A detailed description of the options for Quality of Service can be found in the ISO 8072:1986 specification. The field elements of the structures in use for the option values are self-explanatory. Only the following details remain to be explained.

- If these options are returned with *t_listen*( ), their values are related to the incoming connection and not to the transport endpoint where *t_listen*( ) was issued. To give an example, the value of T_TCO_PROTECTION is the value sent by the calling transport user, and not the value currently effective for the endpoint (that could be retrieved by *t_optmgmt*( ) with the flag T_CURRENT set). The option is not returned at all if the calling user did not specify it. An analogous procedure applies for the other options. See also Chapter 13.
- If, in a call to *t_accept*( ), the called transport user tries to negotiate an option of higher quality than proposed, the option is rejected and the connection establishment fails (see Section 13.3.4 on page 153).
- The values of the QOS options T_TCO_THROUGHPUT, T_TCO_TRANSDEL, T_TCO_RESERRORRATE, T_TCO_TRANSFFAILPROB, T_TCO_ESTFAILPROB, T_TCO_RELFAILPROB, T_TCO_ESTDELAY, T_TCO_RELDELAY and T_TCO_CONNRESIL have a structured format. A user requesting one of these options might leave a field of the structure unspecified by setting it to T_UNSPEC. The transport provider is then free to select an appropriate value for this field. The transport provider may return T_UNSPEC in a field of the structure to the user to indicate that it has not yet decided on a definite value for this field.

  T_UNSPEC is not a legal value for T_TCO_PROTECTION, T_TCO_PRIORITY and T_TCO_EXPD.
- T_TCO_THROUGHPUT and T_TCO_TRANSDEL
  If *avgthrpt* (average throughput) is not defined (both fields set to T_UNSPEC), the transport provider considers that the average throughput has the same values as the maximum throughput (*maxthrpt*). An analogous procedure applies to T_TCO_TRANSDEL.
- The ISO specification ISO 8073:1986 does not differentiate between average and maximum transit delay. Transport providers that support this option adopt the values of the maximum delay as input for the CR TPDU.
- T_TCO_PROTECTION
  This option defines the general level of protection. The symbolic constants in the following list are used to specify the required level of protection:

  | | |
  |---|---|
  | T_NOPROTECT | No protection feature. |
  | T_PASSIVEPROTECT | Protection against passive monitoring. |
  | T_ACTIVEPROTECT | Protection against modification, replay, addition or deletion. |

  Both flags T_PASSIVEPROTECT and T_ACTIVEPROTECT may be set simultaneously but are exclusive with T_NOPROTECT. If the T_ACTIVEPROTECT or T_PASSIVEPROTECT

flags are set, the user may indicate that this is an absolute requirement by also setting the T_ABSREQ flag.

- T_TCO_PRIORITY
  Five priority levels are defined by XTI:

  | | |
  |---|---|
  | T_PRIDFLT | Lower level. |
  | T_PRILOW | Low level. |
  | T_PRIMID | Medium level. |
  | T_PRIHIGH | High level. |
  | T_PRITOP | Higher level. |

- 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.

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.

### *A.2.1.2 Management Options*

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. T_TCO_LTPDU is the only management option supported by an ISO-over-TCP transport provider.

Avoid specifying both QOS parameters and management options at the same time.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_TCO_LTPDU | t_uscalar_t | length in octets | maximum length of TPDU |
| T_TCO_ACKTIME | t_uscalar_t | time in milliseconds | acknowledge time |
| T_TCO_REASTIME | t_uscalar_t | time in seconds | reassignment time |
| T_TCO_PREFCLASS | t_uscalar_t | see text | preferred class |
| T_TCO_ALTCLASS1 | t_uscalar_t | see text | 1st alternative class |
| T_TCO_ALTCLASS2 | t_uscalar_t | see text | 2nd alternative class |
| T_TCO_ALTCLASS3 | t_uscalar_t | see text | 3rd alternative class |
| T_TCO_ALTCLASS4 | t_uscalar_t | see text | 4th alternative class |
| T_TCO_EXTFORM | t_uscalar_t | T_YES/T_NO/T_UNSPEC | extended format |
| T_TCO_FLOWCTRL | t_uscalar_t | T_YES/T_NO/T_UNSPEC | flowctrl |
| T_TCO_CHECKSUM | t_uscalar_t | T_YES/T_NO/T_UNSPEC | checksum |
| T_TCO_NETEXP | t_uscalar_t | T_YES/T_NO/T_UNSPEC | network expedited data |
| T_TCO_NETRECPTCF | t_uscalar_t | T_YES/T_NO/T_UNSPEC | use of network receipt confirmation |

**Table A-2** Management Options

**Absolute Requirements**

A request for any of these options is considered an absolute requirement.

**Further Remarks**

- If these options are returned with *t_listen*( ) their values are related to the incoming connection and not to the transport endpoint where *t_listen*( ) was issued. That means that *t_optmgmt*( ) with the flag T_CURRENT set would usually yield a different result (see Chapter 13).
- For management options that are subject to peer-to-peer negotiation the following holds: If, in a call to *t_accept*( ), the called transport user tries to negotiate an option of higher quality than proposed, the option is rejected and the connection establishment fails (see Section 13.3.4 on page 153).
- A connection-mode transport provider may allow the transport user to select more than one alternative class. The transport user may use the options T_ALTCLASS1, T_ALTCLASS2, etc. to denote the alternatives. A transport provider only supports an implementation-dependent limit of alternatives and ignores the rest.
- The value T_UNSPEC is legal for all options in Table A-2 on page 268. It may be set by the user to indicate that the transport provider is free to choose any appropriate value. If returned by the transport provider, it indicates that the transport provider has not yet decided on a specific value.
- Legal values for the options T_PREFCLASS, T_ALTCLASS1, T_ALTCLASS2, T_ALTCLASS3 and T_ALTCLASS4 are T_CLASS0, T_CLASS1, T_CLASS2, T_CLASS3, T_CLASS4 and T_UNSPEC.
- If a connection has been established, T_TCO_PREFCLASS will be set to the selected value, and T_ALTCLASS1 through T_ALTCLASS4 will be set to T_UNSPEC, if these options are supported.
- *Warning* on the use of T_TCO_LTPDU: Sensible use of this option requires that the application programmer knows about system internals. Careless setting of either a lower or a higher value than the implementation-dependent default may degrade the performance.

  Legal values for an ISO transport provider are T_UNSPEC and multiples of 128 up to $128*(2^{32} - 1)$ or the largest multiple of 128 that will fit in a `t_uscalar_t`. Values other than powers of 2 between $2^7$ and $2^{13}$ are only supported by transport providers that conform to the 1992 update to ISO 8073.

  Legal values for an ISO-over-TCP provider are T_UNSPEC and any power of 2 between 2**7 and 2**11, and 65531.

  The action taken by a transport provider is implementation-dependent if a value is specified which is not exactly as defined in ISO 8073:1986 or its addendums.
- The management options are not independent of one another, and not independent of the options defined in Section A.2.1.1 on page 266. A transport user must take care not to request conflicting values. If conflicts are detected at negotiation time, the negotiation fails according to the rules for absolute requirements (see Chapter 13). Conflicts that cannot be detected at negotiation time will lead to unpredictable results in the course of communication. Usually, conflicts are detected at the time the connection is established.

Some relations that must be obeyed are:

- If T_TCO_EXP is set to T_YES and T_TCO_PREFCLASS is set to T_CLASS2, T_TCO_FLOWCTRL must also be set to T_YES.
- If T_TCO_PREFCLASS is set to T_CLASS0, T_TCO_EXP must be set to T_NO.
- The value in T_TCO_PREFCLASS must not be lower than the value in T_TCO_ALTCLASS1, T_TCO_ALTCLASS2, and so on.
- Depending on the chosen QOS options, further value conflicts might occur.

### A.2.2 Connectionless-mode Service

The protocol level of all subsequent options is ISO_TP (as in Section A.2.1 on page 266).

All options are association-related, that is, they are options with end-to-end significance (see Chapter 13). They may be negotiated in all XTI states but T_UNINIT.

#### *A.2.2.1 Options for Quality of Service*

These options are all defined in the ISO 8072/Add.1:1986 transport service definition (see the ISO Transport references). The definitions are not repeated here. None of these options are supported by an ISO-over-TCP transport provider, since it does not support connectionless-mode.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_TCL_TRANSDEL | `struct rate` | time in milliseconds | transit delay |
| T_TCL_RESERRORRATE | `struct rate` | OPT_RATIO | residual error rate |
| T_TCL_PROTECTION | `t_uscalar_t` | see text | protection |
| T_TCL_PRIORITY | `t_uscalar_t` | see text | priority |

**Table A-3** Options for Quality of Service

**Absolute Requirements**

A request for any of these options is an absolute requirement.

**Further Remarks**

A detailed description of the options for Quality of Service can be found in ISO 8072/Add.1:1986. The field elements of the structures in use for the option values are self-explanatory. Only the following details remain to be explained.

- These options are negotiated only between the local user and the local transport provider.
- The meaning, type of option value, and the range of legal option values are identical for T_TCO_RESERRORRATE and T_TCL_RESERRORRATE, T_TCO_PRIORITY and T_TCL_PRIORITY, T_TCO_PROTECTION and T_TCL_PROTECTION (see Table A-1 on page 266, ISO 8072:1986).
- T_TCL_TRANSDEL and T_TCO_TRANSDEL are different. T_TCL_TRANSDEL specifies the maximum transit delay expected during a datagram transmission. Note that the type of option value is a **struct rate** contrary to the **struct transdel** of T_TCO_TRANSDEL. The range of legal option values for each field of **struct rate** is the same as that of T_TCO_TRANSDEL.

- If these options are returned with *t_rcvudata*( ) their values are related to the received datagram and not to the transport endpoint where *t_rcvudata*( ) was issued. On the other hand, *t_optmgmt*( ) with the flag T_CURRENT set returns the values that are currently effective for outgoing datagrams.
- The function *t_rcvuderr*( ) returns the option value of the data unit previously sent that produced the error.

### *A.2.2.2 Management Options*

This option is a parameter of an ISO transport protocol, according to ISO 8602. It is not included in the ISO transport service definition ISO 8072/Add.1:1986, but is an additional offer by XTI. Transport users wishing to be truly ISO-compliant should thus not adhere to it.

Avoid specifying both QOS parameters and this management option at the same time.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_TCL_CHECKSUM | `t_uscalar_t` | T_YES/T_NO | checksum computation |

**Table A-4** Management Option

#### Absolute Requirements

A request for this option is an absolute requirement.

#### Further Remarks

T_TCL_CHECKSUM

This is the option allows disabling/enabling of the checksum computation. The legal values are T_YES (checksum enabled) and T_NO (checksum disabled).

If this option is returned with *t_rcvudata*( ), its value indicates whether or not a checksum was present in the received datagram.

The advisability of turning off the checksum check is controversial.

## A.3 Functions

*t_accept*( )

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

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

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

*t_bind*( )

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

*t_close*( )

The t_close() call will cause a close() call to be made on the descriptor of this XTI communication endpoint. If there are no other descriptors in this process or any other process which reference this communication endpoint, the close() call will perform an abortive release on any connection associated with this endpoint.

*t_connect*( )

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

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

*t_getinfo*( )

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

— the transport class negotiated during connection establishment (ISO transport provider only)

— the negotiation of expedited data transfer for this connection.

In T_DATAXFER, the *etsdu* field in the **t_info** structure is set to T_INVALID (–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 following table.

| Parameters | Before Call | After Call | | | |
|---|---|---|---|---|---|
| | | **Connection Class 0** | **Connection Class 1-4** | **Connectionless** | **ISO-over-TCP** |
| *fd* | x | / | / | / | / |
| *info→addr* | | x | x | x | x |
| *info→options* | / | x Note 1 | x Note 1 | x Note 1 | x Note 1 |
| *info→tsdu* | / | x Note 2 | x Note 2 | 0→63488 | x Note 2 |
| *info→etsdu* | / | T_INVALID | 16 / T_INVALID see Note 3 | T_INVALID | 16 / T_INVALID see Note 3 |
| *info→connect* | / | T_INVALID | 32 | T_INVALID | 32 / T_INVALID |
| *info→discon* | / | T_INVALID | 64 | T_INVALID | 64 / T_INVALID |
| *info→servtype* | / | T_COTS | T_COTS | T_CLTS | T_COTS |
| *info→flags* | / | 0 | 0 | 0 | 0 |

Note 1: 'x' equals T_INVALID (–2) or an integral number greater than zero.
Note 2: 'x' equals T_INFINITE (–1) or an integral number greater than zero.
Note 3: Depending on the negotiation of expedited data transfer.
For RFC 1006 (ISO over TCP) support, the value of *info→etsdu* depends on the negotiation of expedited data transfer.

*t_listen*( )

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

If the user has set *qlen* greater than 1 (on the call to *t_bind*( )), the user may queue up several connection 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 connection request in a finite time. So if the user queues the connection indications for too long before responding to them, the transport provider initiating the connection will disconnect it.

*t_open*( )

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.

The following table gives the characteristics associated with the different classes.

| **Parameters** | **Before Call** | **After Call** | | | |
|---|---|---|---|---|---|
| | | **Connection Class 0** | **Connection Class 1-4** | **Connectionless** | **ISO-over-TCP** |
| *name* | x | / | / | / | / |
| *oflag* | x | / | / | / | / |
| *info→addr* | / | x | x | x | x |
| *info→options* | / | x Note 1 | x Note 1 | x Note 1 | x Note 1 |
| *info→tsdu* | / | x Note 2 | x Note 2 | 0→63488 | x Note 2 |
| *info→etsdu* | / | T_INVALID | 16 | T_INVALID | 16 / T_INVALID |
| *info→connect* | / | T_INVALID | 32 | T_INVALID | 32 / T_INVALID |
| *info→discon* | / | T_INVALID | 64 | T_INVALID | 64 / T_INVALID |
| *info→servtype* | / | T_COTS | T_COTS | T_CLTS | T_COTS |
| *info→flags* | / | 0 | 0 | 0 | 0 |

Note 1: 'x' equals T_INVALID (–2) or an integral number greater than zero.
Note 2: 'x' equals T_INFINITE (–1) or an integral number greater than zero.

*t_rcv*( ) 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.

*t_rcvconnect*( ) On return, the *call→addr* structure contains the remote calling TSAP. Since, at most, 32 octets of data will be returned to the user, *call→udata.maxlen* should be set to 32 before the call to *t_rcvconnect*( ).

*t_rcvdis*( ) Since, at most, 64 octets of data will be returned to the user, *discon→udata.maxlen* should be set to 64 before the call to *t_rcvdis*( ).

*t_rcvudata*( ) The *unitdata→addr* structure specifies the remote TSAP. 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.

*t_rcvuderr*( ) The *uderr→addr* structure contains the remote TSAP.

*t_snd*( ) Zero byte TSDUs are not supported. The T_EXPEDITED flag is not a legal flag unless expedited data has been negotiated for this connection.

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

*t_sndudata*( ) The *unitdata→addr* structure specifies the remote TSAP. The ISO connectionless-mode transport service does not support the sending of expedited data. This function is not supported by an ISO-over-TCP transport provider.

## A.4 The <xti_osi.h> Header File

This section describes the effects of including the **<xti_osi.h>** header file, which is made available by each XTI implementation that supports use of XTI over OSI, and is included by applications that use the XTI functions for communication via the OSI protocol suite.

The data definitions and macros specified in this section need not be contained in **<xti_osi.h>** itself, but must be available to the application when **<xti.h>** is included and **<xti_osi.h>** is included after **<xti.h>**.

**Note:** Applications written to compilation environments earlier than those required by this issue of the specification (see and defining _XOPEN_SOURCE to be less than 500 may have the data structures and constants of this header automatically exposed by the inclusion of **<xti.h>** for compatibility.

Certain symbols may be exposed to applications including **<xti_osi.h>** for compatibility with applications transitioning from older issues of this specification where their semantics are specified. Exposing these symbols is allowed but not required. Symbols that may be exposed in this implementation-dependent manner are:

T_LTPDUDFLT
ISO_TP
TCO_THROUGHPUT
TCO_TRANSDEL
TCO_RESERRORRATE
TCO_TRANSFFAILPROB
TCO_ESTFAILPROB
TCO_RELFAILPROB
TCO_ESTDELAY
TCO_RELDELAY
TCO_CONNRESIL
TCO_PROTECTION
TCO_PRIORITY
TCO_EXPD
TCL_TRANSDEL
TCL_RESERRORRATE
TCL_PROTECTION
TCL_PRIORITY
TCO_LTPDU
TCO_ACKTIME
TCO_REASTIME
TCO_EXTFORM
TCO_FLOWCTRL
TCO_CHECKSUM
TCO_NETEXP
TCO_NETRECPTCF
TCO_PREFCLASS
TCO_ALTCLASS1
TCO_ALTCLASS2
TCO_ALTCLASS3
TCO_ALTCLASS4
TCL_CHECKSUM

There is an example **<xti_osi.h>** header file in Appendix E on page 317.

**NAME**

**SYNOPSIS**

```
#include <xti_osi.h>
```

**DESCRIPTION**

The following symbol is defined to identify the ISO Transport protocol level:

| Symbol | Protocol Level |
|---|---|
| T_ISO_TP | ISO transport |

The following symbols are defined, with distinct values, to identify the ISO transport classes:

| Symbol | Transport Class |
|---|---|
| T_CLASS0 | transport class 0 |
| T_CLASS1 | transport class 1 |
| T_CLASS2 | transport class 2 |
| T_CLASS3 | transport class 3 |
| T_CLASS4 | transport class 4 |

The following symbols are defined, with distinct values, to identify priorities:

| Symbol | Priority |
|---|---|
| T_PRITOP | top |
| T_PRIHIGH | high |
| T_PRIMID | mid |
| T_PRILOW | low |
| T_PRIDFLT | default |

The following symbols are defined to identify protection levels. Their values are bitwise distinct from each other and from the value of T_ABSREQ:

| Symbol | Protection Level |
|---|---|
| T_NOPROTECT | no protection |
| T_PASSIVEPROTECT | passive protection |
| T_ACTIVEPROTECT | active protection |

Structure type rate is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| targetvalue | t_scalar_t | target value |
| minacceptvalue | t_scalar_t | value of minimum acceptable quality |

Structure type reqvalue is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| called | struct rate | called rate |
| calling | struct rate | calling rate |

Structure type thrpt is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| maxthrpt | struct reqvalue | maximum throughput |
| avgthrpt | struct reqvalue | average throughput |

Structure type transdel is defined with the following members:

| Member | Type | Contents |
|---|---|---|
| maxdel | struct reqvalue | maximum transit delay |
| avgdel | struct reqvalue | average transit delay |

The following symbols are defined with integer values to identify options for quality of service and expedited data defined in ISO 8072:1994. The values of symbols with prefix T_TCO are distinct, and the values of symbols with prefix T_TCL are distinct.

| Symbol | Option |
|---|---|
| T_TCO_THROUGHPUT | connection mode throughput |
| T_TCO_TRANSDEL | connection mode transit delay |
| T_TCO_RESERRORRATE | connection mode residual error rate |
| T_TCO_TRANSFFAILPROB | connection mode transfer failure probability |
| T_TCO_ESTFAILPROB | connection establishment failure probability |
| T_TCO_RELFAILPROB | connection release failure probability |
| T_TCO_ESTDELAY | connection establishment delay |
| T_TCO_RELDELAY | connection release delay |
| T_TCO_CONNRESIL | connection resilience |
| T_TCO_PROTECTION | connection mode protection |
| T_TCO_PRIORITY | connection mode priority |
| T_TCO_EXPD | connection mode expedited data |
| T_TCL_TRANSDEL | connectionless mode transit delay |
| T_TCL_RESERRORRATE | connectionless mode residual error rate |
| T_TCL_PROTECTION | connectionless mode protection |
| T_TCL_PRIORITY | connectionless mode priority |

The following symbols are defined with integer values to identify management options. The values of symbols with prefix T_TCO are distinct.

| Symbol | Option |
|---|---|
| T_TCO_LTPDU | maximum TPDU length |
| T_TCO_ACKTIME | cknowledge time |

| | |
|---|---|
| T_TCO_REASTIME | reassignment time |
| T_TCO_EXTFORM | extended format |
| T_TCO_FLOWCTRL | flow control |
| T_TCO_CHECKSUM | connection mode checksum |
| T_TCO_NETEXP | network expedited data |
| T_TCO_NETRECPTCF | use of network receipt confirmation |
| T_TCO_PREFCLASS | preferred class |
| T_TCO_ALTCLASS1 | alternative class 1 |
| T_TCO_ALTCLASS2 | alternative class 2 |
| T_TCO_ALTCLASS3 | alternative class 3 |
| T_TCO_ALTCLASS4 | alternative class 4 |
| T_TCL_CHECKSUM | connectionless mode checksum |

**SEE ALSO**

**<xti.h>**, **<xti_inet.h>**.

**CHANGE HISTORY**

First released in Issue 5.2.

*Appendix B*

# *Guidelines for Use of XTI*

## B.1 Transport Service Interface Sequence of Functions

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

- It is the responsibility of the transport provider to keep a record of the state of the interface as seen by the transport user.
- The transport provider will not process a function that places the interface out of state.
- If the user issues a function out of sequence, the transport provider will indicate this where possible through an error return on that function. The state will not change. In this case, if any data is passed with the function when not in the T_DATAXFER state, that data will not be accepted or forwarded by the transport provider.
- The uninitialised state (T_UNINIT) of a transport endpoint is the initial state. The endpoint must be initialised and bound before the transport provider may view it as active.
- The uninitialised state is also the final state, and the transport endpoint must be viewed as unused by the transport provider. The *t_close*( ) function will close the transport endpoint and free the transport library resources for another endpoint.
- According to Table 12-5 on page 146, *t_close*( ) should only be issued from the T_UNBND state. If it is issued from any other state, and no other user has that endpoint open, the action will be abortive, the transport endpoint will be successfully closed, and the library resources will be freed for another endpoint. When *t_close*( ) is issued, the transport provider must ensure that the address associated with the specified transport endpoint has been unbound from that endpoint. The provider sends appropriate disconnects if *t_close*( ) is not issued from the unbound state.

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

- The transport connection release phase can be initiated at any time during the connection establishment phase or data transfer phase.
- The only time the state of a transport service interface of a transport endpoint may be transferred to another transport endpoint is when the *t_accept*( ) function specifies such action. The following rules then apply to the cooperating transport endpoints:
  - — The endpoint that is to accept the current state of the interface should either be in state T_UNBND or be in state T_IDLE with the *qlen* paramenter set to 0.

## B.2 Example in Connection-oriented Mode

Figure B-1 on page 281 shows the allowable sequence of functions of an active user and passive user communicating using a connection-mode transport service. This example is not meant to show all the functions that must be called, but rather to highlight the important functions that request a particular service. Blank lines are used to indicate that the function would be called by another user prior to a related function being called by the remote user. For example, the active user calls *t_connect*( ) to request a connection and the passive user would receive an indication of the connection request (via the return from *t_listen*( )) and then would call the *t_accept*( ).

The state diagram in Figure B-1 on page 281 shows the flow of the events through the various states. The active user is represented by a solid line and the passive user is represented by a dashed line. This example shows a successful connection being established and terminated using connection-mode transport service without orderly release. For a detailed description of all possible states and events, see Table 12-7 on page 147.

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

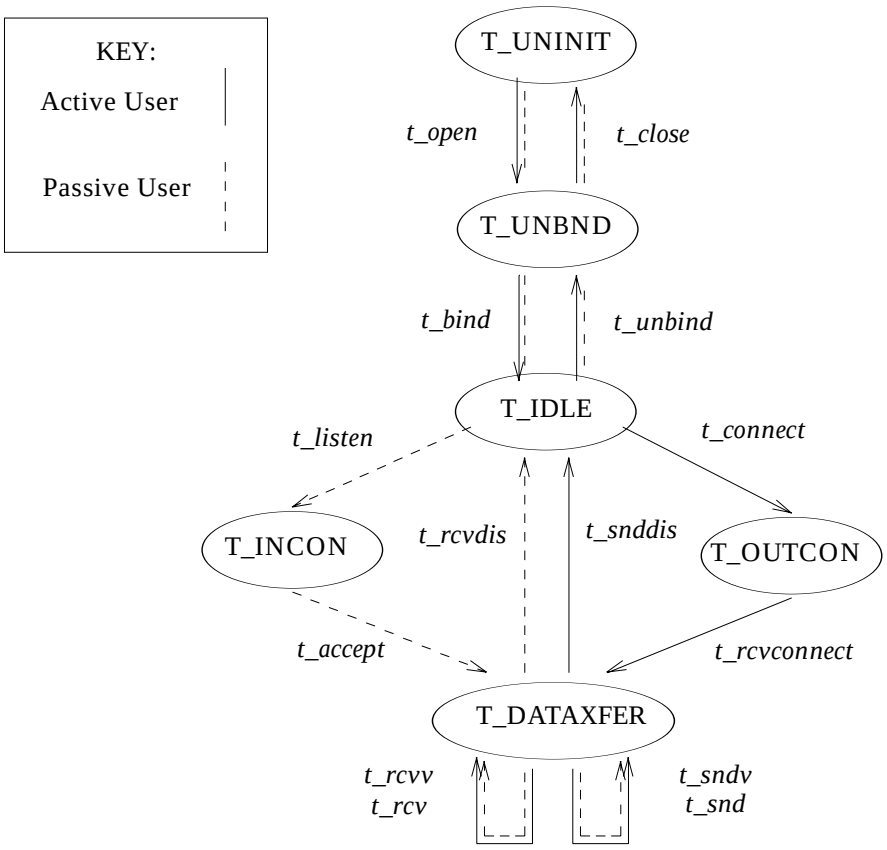


**Figure B-1** Sequence of Transport Functions in Connection-oriented Mode

## B.3 Example in Connectionless Mode

Figure B-2 shows the allowable sequence of functions of user A and user B communicating using a connectionless-mode transport service. This example is not meant to show all the functions that must be called but rather to highlight the important functions that request a particular service. Blank lines are used to indicate that a function would be called by another user prior to a related function being called by the remote user.

The state diagram that follows shows the flow of the events through the various states. This example shows a successful exchange of data between user A and user B. For a detailed description of all possible states and events, see Table 12-7 on page 147.

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

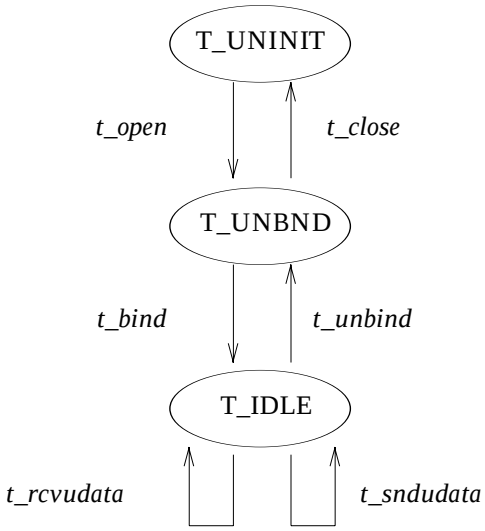


**Figure B-2** Sequence of Transport Functions in Connectionless Mode

## B.4 Writing Protocol-independent Software

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

1. An application should only make use of those functions and mechanisms described as being mandatory features of XTI.
2. In the connection-mode service, the concept of a transport service data unit (TSDU) may not be supported by all transport providers. The user should make no assumptions about the preservation of logical data boundaries across a connection.
3. If an application is not intended to run only over an ISO transport provider, then the name of the device should not be hard-coded into it. While software may be written for a particular class of service (for example, connectionless-mode service), it should not be written to depend on any attribute of the underlying protocol.
4. The protocol-specific service limits returned on the *t_open*( ) and *t_getinfo*( ) functions must not be exceeded. It is the responsibility of the user to access these limits and then adhere to the limits throughout the communication process.
5. The user program should not look at or change options that are specific to the underlying protocol. The *t_optmgmt*( ) function enables a user to access default protocol options from the transport provider, which may then be blindly passed as an argument on the appropriate connection establishment function. Optionally, the user can choose not to pass options as an argument on connection establishment functions.
6. Protocol-specific addressing issues should be hidden from the user program. Similarly, the user must have some way of accessing destination addresses in an invisible manner, such as through a name server. However, the details for doing so are outside the scope of this interface specification.
7. The reason codes associated with *t_rcvdis*( ) are protocol-dependent. The user should not interpret this information if protocol independence is a concern.
8. The error codes associated with *t_rcvuderr*( ) are protocol-dependent. The user should not interpret this information if protocol independence is a concern.
9. The orderly release facility of the connection-mode service (that is, *t_sndrel*( ) and *t_rcvrel*( )) should not be used by programs targeted for multiple protocol environments. This facility is not supported by all connection-based transport protocols. In particular, its use will prevent programs from successfully communicating with ISO open systems.
10. The semantics of expedited data are different across different transport providers (for example, ISO and TCP). An application intended to run over different transport providers should avoid their use.
11. The semantics of closing a connection may be different across transport providers. For example, closing a connection to ISO is abortive while closing a connection to TCP is orderly. A portable application should not assume either facility is available. If the service provider is of type COTS_ORD, a portable application should use *t_sndrel*( ) / *t_rcvrel*( ) prior to calling *t_close*( ).

## B.5 Event Management

In the absence of a standardised Event Management interface, the following guidelines are offered for the use of existing and widely available mechanisms by XTI applications.

These guidelines provide information additional to that given in Section 10.7 on page 124 and Section 10.9 on page 126.

For applications to use XTI in a fully asynchronous manner, they will need to use the facilities of an Event Management (EM) Interface. Such an EM will allow the application to be notified of a number of XTI events over a range of active endpoints. These events may be associated with:

- connection indication
- data indication
- disconnection indication
- flow control being lifted.

In the same way, the EM mechanism should allow the application to be notified of events coming from external sources, such as:

- asynchronous I/O completion
- expiration of timer
- resource availability.

When handling multiple transport connections, the application could either:

- fork a process for each new connection to be handled

  or:

- handle all connections within a single process by making use of the EM facilities.

The application will have to maintain an appropriate balance and choose the right trade-off between the number of processes and the number of connections managed per process in order to minimise the resulting overhead.

Unfortunately, the system facilities to suspend and await notification of an event are presently system-dependent, although work is in progress within standards bodies to provide a unified and portable mechanism.

Hence, for the foreseeable future, applications could use whatever underlying system facilities exist for event notification.

### B.5.1 Short-term Solution

Many vendors currently provide either the System V *poll*( ) or BSD *select*( ) system calls which both give the ability to suspend until there is activity on a member of a set of file descriptors or a timeout.

Given the fact that a transport endpoint identifying a transport connection maps to a file descriptor, applications can take advantage of such EM mechanisms offered by the system (for example, *poll*( ) or *select*( )). The design of more efficient and sophisticated applications, that make full use of all the XTI features, then becomes easily possible.

Guidelines for the use of *poll*( ) and *select*( ) are included in manual-page format, following the end of this section.

### B.5.2 XTI Events

The XTI events can be divided into two classes of events.

- **Class 1**: events related to reception of data.

| | |
|---|---|
| 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 | Disconnection request indication. |
| T_ORDREL | Orderly release request indication. |
| T_UDERR | Notification of an error in a previously sent datagram. |

This class of events should always be monitored by the application.

- **Class 2**: events related to emission of data (flow control).

| | |
|---|---|
| T_GODATA | Normal data may be sent again. |
| T_GOEXDATA | Expedited data may be sent again. |

This class of events informs the application that flow control restrictions have been lifted on a given file descriptor.

The application should request to be notified of this class of events whenever a flow control restriction has previously occurred on this endpoint (for example, [TFLOW] error has been returned on a *t_snd*( ) call).

Note that this class of event should not be monitored systematically otherwise the application would be notified each time a message is sent.

## B.6 The Poll Function

Refer to the description of *poll*( ) in the referenced **XSH** specification. Moreover, Chapter 2 on page 11 of the current document gives additional information on the specific effect of *poll*( ) when applied to Sockets.

### B.6.1 Example of Use of Poll

The following description gives the outline of an XTI server program making use of the *poll*( ) function.

```
/*
 * This is a simple server application example to show how poll( ) can
 * be used in a portable manner to wait for the occurrence of XTI events.
 * In this example, poll( ) is used to wait for the events T_LISTEN,
 * T_DISCONNECT, T_DATA and T_GODATA.
 *
 * A transport endpoint is opened in asynchronous mode over a
 * message-oriented transport provider (for example, ISO). The endpoint
 * is bound with qlen = 1 and the application enters an endless loop
 * to wait for all incoming XTI events on all its active endpoints.
 * For all connection indications received, a new endpoint is opened
 * with qlen = 0 and the connection request is accepted on that endpoint.
 * For all established connections, the application waits for data
 * to be received from one of its clients, sends the received data
 * back to the sender and waits for data again.
 * The cycle repeats until all the connections are released by
 * the clients. The disconnection indications are processed and the
 * endpoints closed.
 *
 * The example references two fictitious functions:
 *
 * -  int get_provider(int tpid, char * tpname)
 *      Given a number as transport provider id, the function returns in
 *      tpname a string as transport provider name that can be used with
 *      t_open( ). This function hides the different naming schemes of
 *      different XTI implementations.
 *
 * -  int get_address(char * symb_name, struct netbuf address)
 *      Given a symbolic name symb_name and a pointer to a struct netbuf
 *      with allocated buffer space as input, the function returns a
 *      protocol address. This function hides the different addressing
 *      schemes of different XTI implementations.
 */

/*
 * General Includes
 */
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <xti.h>

/*
 * Include files for poll( )
 */
#include <poll.h>

/*
 * Various Defines
 */
/*
 * The XTI events T_CONNECT, T_DISCONNECT, T_LISTEN, T_ORDREL and T_UDERR
 * are related to one of the poll flags in INEVENTS (to which one, depends
 * on the implementation). POLLOUT means that (at least) normal data may
 * be sent, and POLLWRBAND that expedited data may be sent.
 */
```

```
#define ERREVENTS     (POLLERR | POLLHUP | POLLNVAL)
#define INEVENTS      (POLLIN | POLLRDNORM | POLLRDBAND | POLLPRI)
#define OUTEVENTS     (POLLOUT | POLLWRBAND)
#define MY_PROVIDER      1   /*  transport provider id            */
#define MAXSIZE       4000   /*  size of send/receive buffer      */
#define TPLEN           30   /*  maximum length of provider name  */
#define MAXCNX          10   /*  maximum number of connections    */


extern int      errno;

/*
 *  Declaration of non-integer external functions
 */
void    exit( );
void    perror( );

/* ============================================================= */

main( )
{

   register int     i;                    /*  loop variable               */
   register int     num;                  /*  return value of t_snd( )    */
                                          /*  and t_rcv( )                */
   int              discflag = 0;         /*  flag to indicate a          */
                                          /*  disc indication             */
   int              errflag = 0;          /*  flag to indicate an error   */
   int              event;                /*  stores events returned      */
                                          /*  by t_look( )                */
   int              fd;                   /*  current file descriptor     */
   int              fdd;                  /*  file descriptor             */
                                          /*  for t_accept( )             */
   int              flags;                /*  used with t_rcv( )          */
   char             *datbuf;              /*  current send/receive buffer */
   unsigned int     act = 0;              /*  active endpoints            */
   struct t_info    info;                 /*  used with t_open( )         */
   struct t_bind    *preq;                /*  used with t_bind( )         */
   struct t_call    *pcall;               /*  used with t_listen( )       */
                                          /*  and t_accept( )             */
   struct t_discon  discon;               /*  used with t_rcvdis( )       */
   char             tpname[TPLEN];        /*  transport provider name     */
   char             buf[MAXCNX][MAXSIZE]; /*  send/receive buffers        */
   int              rcvdata[MAXCNX];      /*  amount of data              */
                                          /*  already received            */
   int              snddata[MAXCNX];      /*  amount of data already sent */

   struct pollfd    fds[MAXCNX];          /*  used with poll( )           */


    /*
     * Get name of transport provider
     */
    if (get_provider(MY_PROVIDER, tpname) == -1) {
            perror(">>> get_provider failed");
            exit(1);
    }

    /*
```

```
 * Establish a transport endpoint in asynchronous mode
 */
if ((fd = t_open(tpname, O_RDWR | O_NONBLOCK, &info)) == -1) {
        t_error(">>> t_open failed");
        exit(1);
}

/*
 * Allocate memory for the parameters passed with t_bind( ).
 */
if ((preq = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR)) == NULL) {
        t_error(">>> t_alloc(T_BIND) failed");
        t_close(fd);
        exit(1);
}

/*
 * Given a symbolic name ("MY_NAME"), get_address returns an address
 * and its length in preq->addr.buf and preq->addr.len.
 */
if (get_address("MY_NAME", &(preq->addr)) == -1) {
        perror(">>> get_address failed");
        t_close(fd);
        exit(1);
}
preq->qlen = 1;         /* is a listening endpoint */

/*
 *  Bind the local protocol address to the transport endpoint.
 *  The returned information is discarded.
 */
if (t_bind(fd, preq, NULL) == -1) {
        t_error(">>> t_bind failed");
        t_close(fd);
        exit(1);
}
if (t_free(preq, T_BIND) == -1) {
        t_error(">>> t_free failed");
        t_close(fd);
        exit(1);
}

/*
 * Allocate memory for the parameters used with t_listen.
 */
if ((pcall = (struct t_call *) t_alloc(fd, T_CALL, T_ALL)) == NULL) {
        t_error(">>> t_alloc(T_CALL) failed");
        t_close(fd);
        exit(1);
}

/*
 * Initialise entry 0 of the fds array to the listening endpoint.
 * To be portable across different XTI implementations,
 * register for INEVENTS and not for POLLIN.
 */
fds[act].fd = fd;
fds[act].events = INEVENTS;
fds[act].revents = 0;
```

```
rcvdata[act] = 0;
snddata[act] = 0;
act = 1;

/*
 * Enter an endless loop to wait for all incoming events.
 * Connect requests are accepted on new opened endpoints.
 * The example assumes that data is first sent by the client.
 * Then, the received data is sent back again and so on, until
 * the client disconnects.
 * Note that the total number of active endpoints (act) should
 * at least be 1, corresponding to the listening endpoint.
 */
fprintf(stderr, "Waiting for XTI events...\n");
while (act > 0) {
    /*
     * Wait for any events
     *
     */
    if (poll(&fds, (size_t)act, (int) -1) == -1) {
    perror(">>> poll failed");
                exit(1);
    }
    /*
     * Process incoming events on all active endpoints
     */
    for (i = 0 ; i < act ; i++) {
        if (fds[i].revents == 0)
            continue;       /* no event for this endpoint */
        if (fds[i].revents & ERREVENTS) {
            fprintf(stderr, "[%d] Unexpected poll events: 0x%x\n",
                            fds[i].fd, fds[i].revents);
            continue;
        }
        /*
         * set the current endpoint
         * set the current send/receive buffer
         */
        fd = fds[i].fd;
        datbuf = buf[i];

        /*
         * Check for events
         */
        switch((event = t_look(fd))) {
        case T_LISTEN:
            /*
             * Must be a connection indication
             */
            if (t_listen(fd, pcall) == -1) {
                t_error(">>> t_listen failed");
                exit(1);
            }
            /*
             * If it will exceed the maximum number
             * of connections that the server can handle,
             * reject the connection indication.
             */
            if (act >= MAXCNX) {
```

```
                fprintf(stderr, ">>> Connection request rejected\n");
                if (t_snddis(fd, pcall) == -1)
                    t_error(">>> t_snddis failed");
                continue;
            }
            /*
             * Establish a transport endpoint
             * in asynchronous mode
             */
            if ((fdd = t_open(tpname, O_RDWR | O_NONBLOCK, &info))
                                                          == -1) {
                t_error(">>> t_open failed");
                continue;
            }
            /*
             * Accept connection on this endpoint.
             * fdd no longer needs to be bound,
             * t_accept( ) will do it.
             */
            if (t_accept(fd, fdd, pcall) == -1) {
                t_error(">>> t_accept failed");
                t_close(fdd);
                continue;
            }
            fprintf(stderr, "Connection [%d] opened\n", fdd);
            /*
             * Register for all flags that might indicate
             * a T_DATA or T_DISCONNECT event, i. e.,
             * register for INEVENTS (to be portable
             * through all XTI implementations).
             */
            fds[act].fd = fdd;
            fds[act].events = INEVENTS;
            fds[act].revents = 0;
            rcvdata[act] = 0;
            snddata[act] = 0;
            act++;
            break;

        case T_DATA:
            /*
             * Must be a data indication
             */
            if ((num = t_rcv(fd, (datbuf + rcvdata[i]),
               (MAXSIZE - rcvdata[i]), &flags)) == -1) {
                switch (t_errno) {
                case TNODATA:
                     /* No data is currently
                      * available: repeat the loop
                      */
                    continue;
                case TLOOK:
                     /* Must be a T_DISCONNECT event:
                      * set discflag
                      */
                    event = t_look(fd);
                    if (event == T_DISCONNECT) {
                        discflag = 1;
```

```
            break;
        }
        else
            fprintf(stderr, "Unexpected event %d\n",
                                                    event);
    default:
        /* Unexpected failure */
        t_error(">>> t_rcv failed");
        fprintf(stderr, "connection id: [%d]\n", fd);
        errflag = 1;
        break;
    }
}

if (discflag || errflag)
                    /* exit from the event switch */
    break;
fprintf(stderr, "[%d] %d bytes received\n", fd, num);
rcvdata[i] += num;
if (rcvdata[i] < MAXSIZE)
    continue;
if (flags & T_MORE) {
    fprintf(stderr, "[%d] TSDU too long for receive
                                      buffer\n", fd);
    errflag = 1;
    break;  /* exit from the event switch */
}

/*
 * Send the data back:
 * Repeat t_snd( ) until either the whole TSDU
 * is sent back, or an event occurs.
 */
fprintf(stderr, "[%d] sending data back\n", fd);
do {
    if ((num = t_snd(fd, (datbuf + snddata[i]),
        (MAXSIZE - snddata[i]), 0)) == -1) {
        switch (t_errno) {
        case TFLOW:
            /*
             * Register for the flags
             * OUTEVENTS to get awaken by
             * T_GODATA, and for INEVENTS
             * to get aware of T_DISCONNECT
             * or T_DATA.
             */
            fds[i].events |= OUTEVENTS;
            continue;

        case TLOOK:
            /*
             * Must be a T_DISCONNECT event:
             * set discflag
             */
            event = t_look(fd);
            if (event == T_DISCONNECT) {
                discflag = 1;
                break;
            }
```

```
                    else
                        fprintf(stderr, "Unexpected event %d\n",
                                                        event);

                default:
                    t_error(">>> t_snd failed");
                    fprintf(stderr, "connection id: [%d]\n", fd);
                    errflag = 1;
                    break;
                }
            }
            else {
                snddata[i] += num;
            }
        } while (MAXSIZE > snddata[i] && !discflag && !errflag);
        /*
         * Reset send/receive counters
         */
        rcvdata[i] = 0;
        snddata[i] = 0;
        break;

    case T_GODATA:
        /*
         * Flow control restriction has been lifted
         * restore initial event flags
         */
        fds[i].events = INEVENTS;
        continue;
    case T_DISCONNECT:
        /*
         * Must be a disconnection indication
         */
        discflag = 1;
        break;
    case -1:
        /*
         * Must be an error
         */
        t_error(">>> t_look failed");
        errflag = 1;
        break;
    default:
        /*
         * Must be an unexpected event
         */
        fprintf(stderr, "[%d] Unexpected event %d\n", fd, event);
        errflag = 1;
        break;
    }       /* end event switch */

    if (discflag) {
        /*
         * T_DISCONNECT has been received.
         * User data is not expected.
         */
        if (t_rcvdis(fd, &discon) == -1)
            t_error(">>> t_rcvdis failed");
        else
```

```
                    fprintf(stderr, "[%d] Disconnection reason: 0x%x\n",
                                                        fd, discon.reason);
                }

                if (discflag || errflag) {
                    /*
                     * Close transport endpoint and
                     * decrement number of active connections
                     */
                    t_close(fd);
                    act--;
                    /* Move last entry of fds array to current slot,
                     * adjust internal counters and flags
                     */
                    fds[i].events = fds[act].events;
                    fds[i].revents = fds[act].revents;
                    fds[i].fd = fds[act].fd;
                    discflag = 0;   /* clear disconnection flag */
                    errflag = 0;    /* clear error flag */
                    i--;    /* Redo the for( ) event loop to consider
                             * events related to the last entry of
                             * fds array */
                    fprintf(stderr, "Connection [%d] closed\n", fd);
                }

            }               /* end of for( ) event loop */

        }       /* end of while( ) loop */
        fprintf(stderr, ">>> Warning: no more active endpoints\n");
        exit(1);
    }
```

## B.7 The Select Function

Refer to the description of *select*( ) in the referenced **XSH** specification. Moreover, Chapter 2 on page 11 of the current document gives additional information on the specific effect of *select*( ) when applied to Sockets.

### B.7.1 Example of Use of Select

The following gives the outline of an XTI server program making use of select().

The following describes the outline of an XTI server program making use of the *select*( ) function.

```
/*
 * This is a simple server application example to show how select( ) can
 * be used in a portable manner to wait for the occurrence of XTI events.
 * In this example, select( ) is used to wait for the events T_LISTEN,
 * T_DISCONNECT, T_DATA and T_GODATA.
 *
 * A transport endpoint is opened in asynchronous mode over a
 * message-oriented transport provider (for example, ISO). The endpoint is
 * bound with qlen = 1, and the application enters an endless loop to wait
 * for all incoming XTI events on all its active endpoints.
 * For all connection indications received, a new endpoint is opened with
 * qlen = 0 and the connection request is accepted on that endpoint.
 * For all established connections, the application waits for data to be
 * received from one of its clients, sends the received data back to the
 * sender and waits for data again.
 * The cycle repeats until all the connections are released by the clients.
 * The disconnection indications are processed and the endpoints closed.
 *
 * The example references two fictitious functions:
 *
 * -  int get_provider(int tpid, char * tpname)
 *      Given a number as transport provider id, the function returns in
 *      tpname a string as transport provider name that can be used with
 *      t_open( ). This function hides the different naming schemes of
 *      different XTI implementations.
 *
 * -  int get_address(char * symb_name, struct netbuf address)
 *      Given a symbolic name symb_name and a pointer to a struct netbuf
 *      with allocated buffer space as input, the function returns a
 *      protocol address. This function hides the different addressing
 *      schemes of different XTI implementations.
 */
/*
 * General Includes
 */
#include <fcntl.h>
#include <stdio.h>
#include <xti.h>
/*
 * Include files for select( ).
 */
#include <sys/select.h>
#include <sys/time.h>
```

```
/*
 * Various Defines
 */

#define MY_PROVIDER      1   /*   transport provider id          */
#define MAXSIZE       4000   /*   size of send/receive buffer    */
#define TPLEN           30   /*   maximum length of provider name */
#define MAXCNX          10   /*   maximum number of connections   */


extern int      errno;

/*
 *  Declaration of non-integer external functions.
 */
void    exit( );
void    perror( );

/* ============================================================ */

main( )
{

   register int     i;                    /*   loop variable              */
   register int     num;                  /*   return value of t_snd( )   */
                                          /*   and t_rcv( )               */

   int              discflag = 0;         /*   flag to indicate a         */
                                          /*   disc indication            */
   int              errflag = 0;          /*   flag to indicate an error  */
   int              event;                /*   stores events returned     */
                                          /*   by t_look( )               */
   int              fd;                   /*   current file descriptor    */
   int              fdd;                  /*   file descriptor            */
                                          /*   for t_accept( )            */
   int              flags;                /*   used with t_rcv( )         */
   char             *datbuf;              /*   current send/receive       */
                                          /*   buffer                     */
   size_t           act = 0;              /*   active endpoints           */
   struct t_info    info;                 /*   used with t_open( )        */
   struct t_bind    *preq;                /*   used with t_bind( )        */
   struct t_call    *pcall;               /*   used with t_listen( )      */
                                          /*   and t_accept( )            */
   struct t_discon  discon;               /*   used with t_rcvdis( )      */
   char             tpname[TPLEN];        /*   transport provider name    */

   int              fds[MAXCNX];          /*   array of file descriptors  */
   char             buf[MAXCNX][MAXSIZE]  /*   send/receive buffers       */
   int              rcvdata[MAXCNX];      /*   amount of data             */
                                          /*   already received           */
   int              snddata[MAXCNX];      /*   amount of data already sent */

   fd_set rfds, wfds, xfds;               /*   file descriptor sets       */
                                          /*   for select( )              */
   fd_set rfdds, wfdds, xfdds;            /*   initial values of          */
```

```
                                        /*   file descriptor sets       */
                                        /*   rfds, wfds and xfds        */

/*
 * Get name of transport provider
 */
if (get_provider(MY_PROVIDER, tpname) == -1) {
    perror(">>> get_provider failed");
    exit(1);
}

/*
 * Establish a transport endpoint in asynchronous mode
 */
if ((fd = t_open(tpname, O_RDWR | O_NONBLOCK, &info)) == -1) {
    t_error(">>> t_open failed");
    exit(1);
}

/*
 * Allocate memory for the parameters passed with t_bind( ).
 */
if ((preq = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR)) == NULL) {
    t_error(">>> t_alloc(T_BIND) failed");
    t_close(fd);
    exit(1);
}

/*
 * Given a symbolic name ("MY_NAME"), get_address returns an address
 * and its length in preq->addr.buf and preq->addr.len.
 */
if (get_address("MY_NAME", &(preq->addr)) == -1) {
    perror(">>> get_address failed");
    t_close(fd);
    exit(1);
}
preq->qlen = 1;         /* is a listening endpoint */

/*
 *  Bind the local protocol address to the transport endpoint.
 *  The returned information is discarded.
 */
if (t_bind(fd, preq, NULL) == -1) {
    t_error(">>> t_bind failed");
    t_close(fd);
    exit(1);
}
if (t_free(preq, T_BIND) == -1) {
    t_error(">>> t_free failed");
    t_close(fd);
    exit(1);
}

/*
 * Allocate memory for the parameters used with t_listen.
 */
if ((pcall = (struct t_call *) t_alloc(fd, T_CALL, T_ALL)) == NULL) {
    t_error(">>> t_alloc(T_CALL) failed");
```

```
    t_close(fd);
    exit(1);
}

/*
 * Initialise listening endpoint in descriptor set.
 * To be portable across different XTI implementations,
 * register for descriptor set rfdds and xfdds
 */
FD_ZERO(&rfdds);
FD_ZERO(&xfdds);
FD_ZERO(&wfdds);
FD_SET(fd, &rfdds);
FD_SET(fd, &xfdds);
fds[act] = fd;
rcvdata[act] = 0;
snddata[act] = 0;
act = 1;

/*
 * Enter an endless loop to wait for all incoming events.
 * Connect requests are accepted on a new opened endpoint.
 * The example assumes that data is first sent by the client.
 * Then, the received data is sent back again and so on, until
 * the client disconnects.
 * Note that the total number of active endpoints (act) should
 * at least be 1, corresponding to the listening endpoint.
 */
fprintf(stderr, "Waiting for XTI events...\n");
while (act > 0) {
    /*
     * Wait for any events
     */

/*
 * Set the mask sets rfds, xfds and wfds to their initial values
 */
rfds = rfdds;
xfds = xfdds;
wfds = wfdds;
if (select(OPEN_MAX, &rfds, &wfds, &xfds,
        (struct timeval *) NULL) == -1) {
    perror(">>> select failed");
    exit(1);
}
/*
 * Process incoming events on all active endpoints
 */
for (i = 0 ; i < act ; i++) {
    /*
     * set the current endpoint
     * set the current send/receive buffer
     */
    fd = fds[i];
    datbuf = buf[i];

    if (FD_ISSET(fd, &xfds)) {
        fprintf(stderr, "[%d] Unexpected select events\n", fd);
        continue;
```

```
}
if (!FD_ISSET(fd, &rfds) && !FD_ISSET(fd, &wfds))
    continue;       /* no event for this endpoint */

/*
 * Check for events
 */
switch((event = t_look(fd))) {
case T_LISTEN:
    /*
     * Must be a connection indication
     */
    if (t_listen(fd, pcall) == -1) {
        t_error(">>> t_listen failed");
        exit(1);
    }

    /*
     * If it will exceed the maximum number
     * of connections that the server can handle,
     * reject the connection indication.
     */
    if (act >= MAXCNX) {
        fprintf(stderr, ">>> Connection request
                                        rejected\n");
        if (t_snddis(fd, pcall) == -1)
            t_error(">>> t_snddis failed");
        continue;
    }
    /*
     * Establish a transport endpoint
     * in asynchronous mode
     */
    if ((fdd = t_open(tpname, O_RDWR | O_NONBLOCK,
                                       &info)) == -1) {
        t_error(">>> t_open failed");
        continue;
    }
    /*
     * Accept connection on this endpoint.
     * fdd no longer needs to be bound,
     * t_accept( ) will do it
     */
    if (t_accept(fd, fdd, pcall) == -1) {
        t_error(">>> t_accept failed");
        t_close(fdd);
        continue;
    }
    fprintf(stderr, "Connection [%d] opened\n", fdd);

    /*
     * Register for all flags that might indicate
     * a T_DATA or T_DISCONNECT event, i. e.,
     * register for rfdds and xfdds (to be portable
     * through all XTI implementations).
     */
    fds[act] = fdd;
    FD_SET(fdd, &rfdds);
    FD_SET(fdd, &xfdds);
```

```
            rcvdata[act] = 0;
            snddata[act] = 0;
            act++;
            break;

        case T_DATA:
             /* Must be a data indication
             */
            if ((num = t_rcv(fd, (datbuf + rcvdata[i]),
               (MAXSIZE - rcvdata[i]), &flags)) == -1) {
                switch (t_errno) {
                case TNODATA:
                     /* No data is currently
                     * available: repeat the loop
                     */
                    continue;
                case TLOOK:
                     /* Must be a T_DISCONNECT event:
                     * set discflag
                     */
                    event = t_look(fd);
                    if (event == T_DISCONNECT) {
                        discflag = 1;
                        break;
                    }
                    else
                        fprintf(stderr, "Unexpected event %d\n", event);

                default:
                    /* Unexpected failure */
                    t_error(">>> t_rcv failed");
                    fprintf(stderr, "connection id: [%d]\n", fd);
                    errflag = 1;
                    break;
                }
            }
            if (discflag || errflag)
                /* exit from the event switch */
                break;
            fprintf(stderr, "[%d] %d bytes received\n", fd, num);
            rcvdata[i] += num;
            if (rcvdata[i] < MAXSIZE)
                continue;
            if (flags & T_MORE) {
                fprintf(stderr, "[%d] TSDU too long for receive
                                                    buffer\n", fd);
                errflag = 1;
                break;  /* exit from the event switch */
            }

            /*
             * Send the data back.
             * Repeat t_snd( ) until either the whole TSDU
             * is sent back, or an event occurs.
             */
            fprintf(stderr, "[%d] sending data back\n", fd);
            do {
                if ((num = t_snd(fd, (datbuf + snddata[i]),
```

```
                (MAXSIZE - snddata[i]), 0)) == -1) {
                switch (t_errno) {
                case TFLOW:
                    /*
                     * Register for wfds to get
                     * awaken by T_GODATA, and for
                     * rfds and xfds to get aware of
                     * T_DISCONNECT or T_DATA.
                     */
                    FD_SET(fd, &wfdds);
                    continue;

                case TLOOK:
                    /*
                     * Must be a T_DISCONNECT event:
                     * set discflag
                     */
                    event = t_look(fd);
                    if (event == T_DISCONNECT) {
                        discflag = 1;
                        break;
                    }
                    else
                        fprintf(stderr, "Unexpected event
                                            %d\n", event);

                default:
                    t_error(">>> t_snd failed");
                    fprintf(stderr, "connection id: [%d]\n", fd);
                    errflag = 1;
                    break;
                }
            }
            else {
                snddata[i] += num;
            }
        } while (MAXSIZE > snddata[i] && !discflag && !errflag);
        /*
         * Reset send/receive counter
         */
        rcvdata[i] = 0;
        snddata[i] = 0;
        break;

    case T_GODATA:
        /*
         * Flow control restriction has been lifted
         * restore initial event flags
         */
        FD_CLR(fd, &wfdds);
        continue;
```

```
case T_DISCONNECT:
    /*
     * Must be a disconnection indication
     */
    discflag = 1;
    break;
case -1:
    /*
     * Must be an error
     */
    t_error(">>> t_look failed");
    errflag = 1;
    break;
default:
    /*
     * Must be an unexpected event
     */
    fprintf(stderr, "[%d] Unexpected event %d\n", fd, event);
    errflag = 1;
    break;
}       /* end event switch */

if (discflag) {
    /*
     * T_DISCONNECT has been received.
     * User data is not expected.
     */
    if (t_rcvdis(fd, &discon) == -1)
        t_error(">>> t_rcvdis failed");
    else
        fprintf(stderr, "[%d] Disconnection reason: 0x%x\n",
                                        fd, discon.reason);
    }

    if (discflag || errflag) {
        /*
         * Close transport endpoint and
         * decrement number of active connections
         */
        t_close(fd);
        act--;
        /*
         * Unregister fd from initial mask sets
         */
        FD_CLR(fd, &rfdds);
        FD_CLR(fd, &xfdds);
        FD_CLR(fd, &wfdds);
        /* Move last entry of fds array to current slot,
         * adjust internal counters and flags
         */
        fds[i] = fds[act];
        discflag = 0;   /* clear disconnection flag */
        errflag = 0;    /* clear error flag */
```

```
                i--;    /* Redo the for( ) event loop to consider
                         * events related to the last entry of
                         * fds array */
                fprintf(stderr, "Connection [%d] closed\n", fd);
            }

        }       /* end of for( ) event loop */

    }       /* end of while( ) loop */
    fprintf(stderr, ">>> Warning: no more active endpoints\n");
    exit(1);
}
```

*Appendix C*

# *Use of XTI to Access NetBIOS*

## C.1 Introduction

NetBIOS represents an important *de facto* standard for networking DOS and OS/2 PCs. The X/Open Specification **Protocols for X/Open PC Interworking: SMB** (see the referenced **NetBIOS** specification) provides mappings of NetBIOS services to OSI and IPS transport protocols[17].

XTI defines a transport service interface that is independent of any specific transport provider.

This Appendix defines a standard for using XTI to access NetBIOS transport providers. Applications that use XTI to access NetBIOS transport providers are referred to as ''transport users''.

This Appendix also defines data structures and constants required for NetBIOS transport providers which are exposed through **<xti_netbios.h>** header file.

**Note:** Applications written to compilation environments earlier than those required by this issue of the specification (see Section 1.3 on page 3) and defining _XOPEN_SOURCE to be less than 500, may have these data structures and constants exposed through the inclusion of **<xti.h>**

## C.2 Objectives

The objectives of this standardisation are:

1. to facilitate the development and portability of applications that interwork with the large installed base of NetBIOS applications in a Local Area Network (LAN) environment
2. to enable a single application to use the same XTI interface to communicate with remote applications through either an IPS profile, an OSI profile or a NetBIOS profile (that is, RFC 1001/1002 or TOP/NetBIOS)
3. to provide a common interface that can be used for IPC with clients using either (PC)NFS or SMB protocols for resources sharing.

This Appendix provides a migration step to users moving from proprietary systems in a NetBIOS environment an ''open systems'' environment.

17. The mappings are defined by the Specification of NetBIOS Interface and Name Service Support by Lower Layer OSI Protocols, and RFC 1001/RFC 1002 respectively. See the referenced **NetBIOS** specification. The relevant chapters are Chapter 13, NetBIOS Interface to ISO Transport Services, Chapter 14, Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods and Chapter 15, Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Detailed Specification.

## C.3 Scope

No extensions are made to XTI by the definitions provided in this Appendix. This NetBIOS specification is concerned only with standardisation of the mapping of XTI to the NetBIOS facilities, and not a new definition of XTI itself.

This NetBIOS specification applies only to the use of XTI in the single NetBIOS subnetwork case, and does not provide for the support of applications operating in multiple, non-overlapping NetBIOS name spaces.

The following NetBIOS facilities found in various NetBIOS implementations are considered outside the scope of XTI (note that this list is not necessarily definitive):

- LAN.STATUS.ALERT
- RESET
- SESSION STATUS
- TRACE
- UNLINK
- RPL (Remote Program Load)
- ADAPTER STATUS
- FIND NAME
- SEND.NOACK
- CHAIN.SEND.NOACK
- CANCEL
- receiving a datagram on any name
- receiving data on any connection.

It must also be noted that not all commands are specified in the protocols.

Omitting these does not restrict interoperability with the majority of NetBIOS implementations, since they have local significance only (RESET, SESSION STATUS), are concerned with systems management (UNLINK, RPL, ADAPTER STATUS), or are LAN- and vendor-specific (FIND NAME). If and how these functions are made available to the programmer is left to the implementor of this particular XTI implementation.

## C.4 Issues

The primary issues for XTI as a transport interface to NetBIOS concern the passing of NetBIOS names and name type information through XTI, specification of restrictions on XTI functions in the NetBIOS environment, and handling the highly dynamic assignment of NetBIOS names.

## C.5 NetBIOS Names and Addresses

NetBIOS uses 16-octet alphanumeric names as ‘‘transport’’ addresses. NetBIOS names must be exactly 16 octets, with shorter names padded with spaces to 16 octets. In addition, NetBIOS names are either unique names, group names or local names, and must be identified as such in certain circumstances. A local NetBIOS name is a name that is not defended on the network.

The following restrictions should be applied to NetBIOS names. Failure to observe these restrictions may result in unpredictable results.

1. Byte 0 of the name is not allowed to be hexadecimal 00 (0x00).
2. Byte 0 of the name is not allowed to be an asterisk, except as noted elsewhere in this specification to support broadcast datagrams.
3. Names should not begin with company names or trademarks.
4. Names should not begin with hexadecimal FF (0xFF).
5. Byte 15 of the name should not be in the range 0x00 – 0x1F.

The concept of a permanent node name, as provided in the native NetBIOS environment, is not supported in The Open Group’s concept of an ‘‘open system’’.

The following definitions are supplied with any implementation of XTI on top of NetBIOS. These definitions are exposed by the inclusion of **<xti_netbios.h>**.

```
#define T_NB_UNIQUE        0
#define T_NB_GROUP         1
#define T_NB_LOCAL         2
#define T_NB_NAMELEN      16
#define T_NB_BCAST_NAME    "*     "    /* asterisk plus 15 spaces */
```

The protocol addresses passed in calls to *t_bind*( ), *t_connect*( ), etc., are structured as follows:

```
1    2                                17
+----+--------------------------------+
|Type|       NetBIOS Name              |
+----+--------------------------------+
```

*Type* The first octet specifies the type of the NetBIOS name. It may be set to T_NB_UNIQUE, T_NB_GROUP or T_NB_LOCAL.

*NetBIOS Name* Octets 2 through 17 contain the 16-octet NetBIOS name.

All NetBIOS names, complete with the name type identifier, are passed through XTI in a *netbuf* address structure (that is, **struct netbuf addr**), where *addr.buf* points to a NetBIOS protocol address as defined above. This applies to all XTI functions that pass or return a (NetBIOS) protocol address (for example, *t_bind*( ), *t_connect*( ), *t_rcvudata*( ), etc.).

Note, however, that only the *t_bind*( ) and *t_getprotaddr*( ) functions use the name type information. All other functions ignore it.

If the NetBIOS protocol address is returned, the name type information is to be ignored since the NetBIOS transport providers do not provide the type information in the connection establishment phase.

NetBIOS names can become invalid even after they have been registered successfully due to the NetBIOS name conflict resolution process (for example, Top / NetBIOS NameConflictAdvise indication). For existing NetBIOS connections this has no effect since the connection endpoint can still be identified by the *fd*. However, in the connection establishment phase *2t_listen*( ) and *t_connect*( )) this event is indicated by setting *t_errno* to [TBADF].

## C.6 NetBIOS Connection Release

Native NetBIOS implementations provide a linger-on-close release mechanism whereby a transport disconnection request (NetBIOS HANGUP) will not complete until all outstanding send commands have completed. NetBIOS attempts to deliver all queued data by delaying, if necessary, disconnection for a period of time. The period of time might be configurable; a value of 20 seconds is common practice. Data still queued after this time period may get discarded so that delivery cannot be guaranteed.

XTI, however, offers two different modes to release a connection: an abortive mode via *t_snddis*( ) / *t_rcvdis*( ), and a graceful mode via *t_sndrel*( ) / *t_rcvrel*( ). If a connection release is initiated by a *t_snddis*( ), queued send data may be discarded. Only the use of *t_sndrel*( ) guarantees that the linger-on-close mechanism is enabled as described above. The support of *t_sndrel*( ) / *t_rcvrel*( ) is optional and only provided by implementations with servtype T_COTS_ORD (see *t_getinfo*( ) in Section C.8 on page 309).

A call to *t_sndrel*( ) initiates the linger-on-close mechanism and immediately returns with the XTI state changed to T_OUTREL. The NetBIOS provider sends all outstanding data followed by a NetBIOS Close Request. After receipt of a NetBIOS Close Response, the NetBIOS provider informs the transport user, via the event T_ORDREL, that is to be consumed by calling *t_rcvrel*( ). If a timeout occurs, however, a T_DISCONNECT event with a corresponding reason code is generated.

Receive data arriving before the NetBIOS Close Request is sent is indicated by T_DATA and can be read by the transport user.

Calling *t_snddis*( ) initiates an abortive connection release and immediately returns with the XTI state changed to T_IDLE. Outstanding send and receive data may be discarded. The NetBIOS provider sends as many outstanding data as possible prior to closing the connection, but discards any receive data. Some outstanding data may be discarded by the *t_snddis*( ) mechanism, so that not all data can be sent by the NetBIOS provider. Furthermore, an occurring timeout condition could not be indicated to the transport user.

An incoming connection release will always result in a T_DISCONNECT event, never in a T_ORDREL event. To be precise, if the NetBIOS provider receives a Close Request, it discards any pending send and receive data, sends a Close Response and informs the transport user via T_DISCONNECT.

## C.7 Options

No NetBIOS-specific options are defined. An implementation may, however, provide XTI-level options (see *t_optmgmt*( ) on page 195.

## C.8 XTI Functions

*t_accept*( ) No user data may be returned to the caller (call→udata.len=0).

This function may only be used with connection-mode transport endpoints. The *t_accept*( ) function will fail if a user attempts to accept a connection request on a connectionless-mode endpoint and *t_errno* will be set to [TNOTSUPPORT].

*t_alloc*( ) No special considerations for NetBIOS transport providers.

*t_bind*( ) The NetBIOS name and name type values are passed to the transport provider in the *req* parameter (req→addr.buf) and the actual bound address is returned in the *ret* parameter (ret→addr.buf), as described earlier in Section C.5 on page 307. If the NetBIOS transport provider is unable to register the name specified in the *req* parameter, the call to *t_bind*( ) will fail with *t_errno* set to [TADDRBUSY] if the name is already in use, or to [TBADADDR] if it was an illegal NetBIOS name. If the NetBIOS name type is T_NB_LOCAL the name is not defended on the network, that is, it is not registered.

If the *req* parameter is a null pointer or req→addr.len=0, the transport provider may assign an address for the user. This may be useful for outgoing connections on which the name of the caller is not important.

If the name specified in *req* parameter is T_NB_BCAST_NAME, *qlen* must be zero, and the transport endpoint the name is bound to is enabled to receive broadcast datagrams. In this case, the transport endpoint must support connectionless-mode service, otherwise the *t_bind*( ) function will fail and *t_errno* will be set to [TBADADDR].

*t_close*( ) No special considerations for NetBIOS transport providers.

It is assumed that the NetBIOS transport provider will release the NetBIOS name associated with the closed endpoint if this is the only endpoint bound to this name and the name has not already been released as the result of a previous *t_unbind*( ) call on this endpoint.

*t_connect*( ) The NetBIOS name of the destination transport user is provided in the *sndcall* parameter (sndcall→addr.buf), and the NetBIOS name of the responding transport user is returned in the *rcvcall* parameter (rcvcall→addr.buf), as described in Section C.5 on page 307. If the connection is successful, the NetBIOS name of the responding transport user will always be the same as that specified in the *sndcall* parameter.

Local NetBIOS connections are supported. NetBIOS datagrams are sent, if applicable, to local names as well as remote names. No user data may be sent during connection establishment (udata.len=0 in *sndcall*).

This function may only be used with connection-mode transport endpoints. The *t_connect*( ) function will fail if a user attempts to initiate a connection on a connectionless-mode endpoint and *t_errno* will be set to [TNOTSUPPORT].

[TBADF] may be returned in the case that the NetBIOS name associated with the *fd* referenced in the *t_connect*( ) call is no longer in the system name table (see Section C.5 on page 307.

*t_error*( ) No special considerations for NetBIOS transport providers.

*t_free*( ) No special considerations for NetBIOS transport providers.

*t_getinfo*( ) The values of the parameters in the *t_info* structure will reflect NetBIOS transport limitations, as follows:

addr *sizeof*( ) the NetBIOS protocol address, as defined in Section C.5 on page 307.

options Equals –2, indicating no user-settable options.

tsdu Equals the size returned by the transport provider. If the *fd* is associated with a connection-mode endpoint it is a positive value, not larger than 131070. If the *fd* is associated with a connectionless-mode endpoint it is a positive value not larger than 65535[18].

etsdu Equals –2, indicating expedited data is not supported.

connect Equals –2, indicating data cannot be transferred during connection establishment.

discon Equals –2, indicating data cannot be transferred during connection release.

servtype Set to *T_COTS* if the *fd* is associated with a connection-mode endpoint, or *T_CLTS* if associated with a connectionless-mode endpoint. Optionally, may be set to *T_COTS_ORD* if the *fd* is associated with a connection-mode endpoint and the transport provider supports the use of *t_sndrel*( ) / *t_rcvrel*( ) as described in Section C.6 on page 308.

flags Equals T_SNDZERO, indicating that zero TSDUs may be sent.

*t_getprotaddr*( ) The NetBIOS name and name type of the transport endpoint referred to by the *fd* are passed in the *boundaddr* parameter (boundaddr→addr.buf), as described in Section C.5 on page 307; 0 is returned in boundaddr→addr.len if the transport endpoint is in the T_UNBND state. The NetBIOS name currently connected to *fd*, if any, is passed in the *peeraddr* parameter (peeraddr→addr.buf); the value 0 is returned in peeraddr→addr.len if the transport endpoint is not in the T_DATAXFER state.

*t_getstate*( ) No special considerations for NetBIOS transport providers.

*t_listen*( ) On return, the *call* parameter provides the NetBIOS name of the calling transport user (that issued the connection request), as described in Section C.5 on page 307.

No user data may be transferred during connection establishment (call→udata.len=0 on return).

18. For the mappings to OSI and IPS protocols, the value cannot exceed 512 or 1064 respectively.

This function may only be used with connection-mode transport endpoints. The *t_listen*( ) function will fail if a user attempts to *listen* on a connectionless-mode endpoint and *t_errno* will be set to [TNOTSUPPORT]. [TBADF] may be returned in the case that the NetBIOS name associated with the *fd* referenced in the *t_listen*( ) function is no longer in the system name table, as may occur as a result of the NetBIOS name conflict resolution process (for example, TOP/NetBIOS NameConflictAdvise indication).

*t_look*( ) Since expedited data is not supported in NetBIOS, the T_EXDATA and T_GOEXDATA events cannot be returned.

*t_open*( ) No special considerations for NetBIOS transport providers, other than restrictions on the values returned in the *t_info* structure. These restrictions are described in *t_getinfo*( ) on page 182.

*t_optmgmt*( ) No special considerations for NetBIOS transport providers.

*t_rcv*( ) This function may only be used with connection-mode transport endpoints. The *t_rcv*( ) function will fail if a user attempts a receive on a connectionless-mode endpoint and *t_errno* will be set to [TNOTSUPPORT].

The *flags* parameter will never be set to T_EXPEDITED, as expedited data is not supported.

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.

*t_rcvconnect*( ) The NetBIOS name of the transport user responding to the previous connection request is provided in the *call* parameter (call→addr.buf), as described in Section C.5 on page 307.

No user data may be returned to the caller (call→udata.len=0 on return).

This function may only be used with connection-mode transport endpoints. The *t_rcvconnect*( ) function will fail if a user attempts to establish a connection on a connectionless-mode endpoint and *t_errno* will be set to [TNOTSUPPORT].

*t_rcvdis*( ) The following disconnection reason codes are valid for any implementation of a NetBIOS provider under XTI:

```
#define T_NB_ABORT     0x18  /* session ended abnormally */
#define T_NB_CLOSED    0x0A  /* session closed           */
#define T_NB_NOANSWER  0x14  /* no answer (cannot find   */
                             /* name called              */
#define T_NB_OPREJ     0x12  /* session open rejected    */
```

These definitions are exposed by the inclusion of **<xti_netbios.h>**.

*t_rcvrel*( ) As described in Section C.6 on page 308, 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 *t_rcvrel*( ) if it previously initiated a connection release by calling *t_sndrel*( ). As a side effect, the state T_INREL is unreachable for the transport user.

If T_COTS_ORD is not supported by the underlying NetBIOS transport provider, this function will fail with *t_errno* set to [TNOTSUPPORT].

*t_rcvudata*( ) The NetBIOS name of the sending transport user is provided in the *unitdata* parameter (unitdata→addr.buf), as described in Section C.5 on page 307.

The *fd* associated with the *t_rcvudata*( ) function must refer to a connectionless-mode transport endpoint. The function will fail if a user attempts to receive on a connection-mode endpoint and *t_errno* will be set to [TNOTSUPPORT]. [TBADF] may be returned in the case that the NetBIOS name associated with the *fd* referenced in the *t_rcvudata*( ) function is no longer in the system name table, as may occur as a result of the NetBIOS name conflict resolution process (for example, TOP/NetBIOS NameConflictAdvise indication).

To receive a broadcast datagram, the endpoint must be bound to the NetBIOS name T_NB_BCAST_NAME.

*t_rcvuderr*( ) If attempted on a connectionless-mode transport endpoint, this function will fail with *t_errno* set to [TNOUDERR], as no NetBIOS unit data error codes are defined. If attempted on a connection-mode transport endpoint, this function will fail with *t_errno* set to [TNOTSUPPORT].

*t_snd*( ) The T_EXPEDITED flag may not be set, as NetBIOS does not support expedited data transfer.

This function may only be used with connection-mode transport endpoints. The *t_snd*( ) function will fail if a user attempts a send on a connectionless-mode endpoint and *t_errno* will be set to [TNOTSUPPORT].

The maximum value of the *nbytes* parameter is determined by the maximum TSDU size allowed by the transport provider. The maximum TSDU size can be obtained from the *t_getinfo*( ) call.

Data transfer in the NetBIOS environment is record-oriented. The transport user can use the *T_MORE* flag in order to fragment a TSDU and send it via multiple calls to *t_snd*( ). See *t_snd*( ) on page 219 for more details.

NetBIOS does not support the notion of expedited data. A call to *t_snd*( ) with the *T_EXPEDITED* flag will fail with *t_errno* set to [TBADDATA].

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 *t_snd*( ). The actions that are taken are implementation-dependent:

- *t_snd*( ) might fail with *t_errno* set to [TPROTO]
- *t_snd*( ) 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.

*t_snddis*( ) The *t_snddis*( ) function initiates an abortive connection release. The function returns immediately. Outstanding send and receive data may be discarded. See Section C.6 on page 308 for further details.

No user data may be sent in the disconnection request (call→udata.len=0).

This function may only be used with connection-mode transport endpoints. The *t_snddis*( ) function will fail if a user attempts a disconnection request on a

connectionless-mode endpoint and *t_errno* will be set to [TNOTSUPPORT].

*t_sndrel*( ) The *t_sndrel*( ) 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 a T_DISCONNECT event with an appropriate disconnection reason code. See Section C.6 on page 308 for further details.

If the NetBIOS transport provider did not return T_COTS_ORD with *t_open*( ), this function will fail with *t_errno* set to [TNOTSUPPORT].

*t_sndudata*( ) The NetBIOS name of the destination transport user is provided in the *unitdata* parameter (unitdata→addr.buf), as described in Section C.5 on page 307.

The *fd* associated with the *t_sndudata*( ) function must refer to a connectionless-mode transport endpoint. The function will fail if a user attempts this function on a connection-mode endpoint and *t_errno* will be set to [TNOTSUPPORT]. [TBADF] may be returned in the case that the NetBIOS name associated with the *fd* referenced in the *t_sndudata*( ) function is no longer in the system name table, as may occur as a result of the NetBIOS name conflict resolution process (for example, TOP/NetBIOS NameConflictAdvise indication).

To send a broadcast datagram, the NetBIOS name in the NetBIOS address structure provided in *unitdata→addr.buf* must be T_NB_BCAST_NAME.

*t_strerror*( ) No special considerations for NetBIOS transport providers.

*t_sync*( ) No special considerations for NetBIOS transport providers.

*t_unbind*( ) No special considerations for NetBIOS transport providers.

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.

## C.9 Compatibility.

Certain symbols may be exposed to applications including **<xti_netbios.h>** for compatibility with applications transitioning from older issues of this specification where their semantics are specified. Exposing these symbols is allowed but not required. Symbols that may be exposed in this implementation-dependent manner are:

NB_UNIQUE, NB_GROUP, NB_LOCAL, NB_NAMELEN, NB_BCAST_NAME NB_ABORT, NB_CLOSED, NB_NOANSWER, NB_OPREJ

*Appendix D*

# *XTI and TLI*

XTI is based on the System V Interface Definitions (SVID) Issue 2, Volume III, Networking Services Extensions (see Referenced Documents).

XTI provides refinement of the Transport Level Interface (TLI) where such refinement is considered necessary. This refinement takes the form of:

- additional commentary or explanatory text, in cases where the TLI text is either ambiguous or not sufficiently detailed
- modifications to the interface, to cater for service and protocol problems which have been fully considered. In this case, it must be emphasised that such modifications are kept to an absolute minimum, and are intended to avoid any fundamental changes to the interface defined by TLI
- the removal of dependencies on specific UNIX versions and specific transport providers.

## D.1 Restrictions Concerning the Use of XTI

It is important to bear in mind the following points when considering the use of XTI:

- It was stated that XTI ''recommends'' a subset of the total set of functions and facilities defined in TLI, and also that XTI introduces modifications to some of these functions and/or facilities where this is considered essential. For these reasons, an application which is written in conformance to XTI may not be immediately portable to work over a provider which has been written in conformance to TLI.
- XTI does not address management aspects of the interface, that is:
  - — how addressing may be done in such a way that an application is truly portable
  - — no selection and/or negotiation of service and protocol characteristics.

  For addressing, the same is also true for TLI. In this case, it is envisaged that addresses will be managed by a higher-level directory function. For options selection and/or negotiation, XTI attempts to define a basic mechanism by which such information may be passed across the transport service interface, although again, this selection/negotiation may be done by a higher-level management function (rather than directly by the user). Since address structure is not currently defined, the user protocol address is system-dependent.

## D.2 Relationship between XTI and TLI

The following features can be considered as XTI extensions to the System V Release 3 version of TLI:

- Some functions may return more error types. The use of the [TOUTSTATE] error is generalised to almost all protocol functions.
- The transport provider identifier has been generalised to remove the dependence on a device driver implementation.
- Additional events have been defined to help applications make full use of the asynchronous features of the interface.
- Additional features have been introduced to *t_snd*( ), *t_sndrel*( ) and *t_rcvrel*( ) to allow fuller use of TCP transport providers.
- Usage of options for certain types of transport service has been defined to increase application portability.
- Because most XTI functions require read/write access to the transport provider, the usage of flags O_RDONLY and O_WRONLY has been withdrawn from the XTI.
- XTI checks the value of *qlen* and prevents an application from waiting forever when issuing *t_listen*( ).
- XTI allows an application to call *t_accept*( ) with a *resfd* which is not bound to a local address.
- XTI provides the additional utility functions *t_strerror*( ) and *t_getprotaddr*( ).

*Appendix E*

# *Example XTI Header Files*

Section 14.1 on page 161 contains a normative requirement that the contents and structures found in this appendix appear in the **<xti.h>** header.

This Appendix contains example **<xti.h>**, **<xti_osi.h>** and **<xti_inet.h>** headers that satisfy the requirements of Chapter 15 on page 241, Section A.4 on page 275 and Section 16.5 on page 260 The specifications in Chapter 15, Section A.4 and Section 16.5 on page 260 are normative, while the material in this Appendix is informative only. Should there be any conflict between them, the definitions in Chapter 15, Section A.4 on page 275 and Section 16.5 on page 260 take precedence.

## E.1 Example <xti.h> Header

```
#define  t_scalar_t      int
#define  t_uscalar_t     unsigned int


/*
 * The following are the error codes needed by both the kernel
 * level transport providers and the user level library.
 */

#define TBADADDR         1  /* incorrect addr format */
#define TBADOPT          2  /* incorrect option format */
#define TACCES           3  /* incorrect permissions */
#define TBADF            4  /* illegal transport fd */
#define TNOADDR          5  /* couldn't allocate addr */
#define TOUTSTATE        6  /* out of state */
#define TBADSEQ          7  /* bad call sequence number */
#define TSYSERR          8  /* system error */
#define TLOOK            9  /* event requires attention */
#define TBADDATA        10  /* illegal amount of data */
#define TBUFOVFLW       11  /* buffer not large enough */
#define TFLOW           12  /* flow control */
#define TNODATA         13  /* no data */
#define TNODIS          14  /* discon_ind not found on queue */
#define TNOUDERR        15  /* unitdata error not found */
#define TBADFLAG        16  /* bad flags */
#define TNOREL          17  /* no ord rel found on queue */
#define TNOTSUPPORT     18  /* primitive/action not supported */
#define TSTATECHNG      19  /* state is in process of changing */
#define TNOSTRUCTYPE    20  /* unsupported struct-type requested */
#define TBADNAME        21  /* invalid transport provider name */
#define TBADQLEN        22  /* qlen is zero */
#define TADDRBUSY       23  /* address in use */
#define TINDOUT         24  /* outstanding connection indications */
#define TPROVMISMATCH   25  /* transport provider mismatch */
#define TRESQLEN        26  /* resfd specified to accept w/qlen >0 */
#define TRESADDR        27  /* resfd not bound to same addr as fd */
#define TQFULL          28  /* incoming connection queue full */
```

```
#define TPROTO          29   /* XTI protocol error */

/*
 * The following are the events returned.
 */

#define T_LISTEN       0x0001   /* connection indication received */
#define T_CONNECT      0x0002   /* connection confirmation received */
#define T_DATA         0x0004   /* normal data received */
#define T_EXDATA       0x0008   /* expedited data received */
#define T_DISCONNECT   0x0010   /* disconnection received */
#define T_UDERR        0x0040   /* datagram error indication */
#define T_ORDREL       0x0080   /* orderly release indication */
#define T_GODATA       0x0100   /* sending normal data is again possible */
#define T_GOEXDATA     0x0200   /* sending expedited data is again */
                                /* possible */

/*
 * The following are the flag definitions needed by the
 * user level library routines.
 */

#define T_MORE          0x001   /* more data */
#define T_EXPEDITED     0x002   /* expedited data */
#define T_PUSH          0x004   /* send data immediately */
#define T_NEGOTIATE     0x004   /* set opts */
#define T_CHECK         0x008   /* check opts */
#define T_DEFAULT       0x010   /* get default opts */
#define T_SUCCESS       0x020   /* successful */
#define T_FAILURE       0x040   /* failure */
#define T_CURRENT       0x080   /* get current options */
#define T_PARTSUCCESS   0x100   /* partial success */
#define T_READONLY      0x200   /* read-only */
#define T_NOTSUPPORT    0x400   /* not supported */

/*
 *  XTI error return.
 */

extern int t_errno;

/* t_errno is a modifiable lvalue of type int                          */
/* The above definition is typical of a single-threaded environment.   */
/* In a multi-threading environment a typical definition of t_errno is:*/

/* extern int *_t_errno(void);      */
/* #define t_errno (*(_t_errno()))  */



/*
 * iov maximum
 */
#define T_IOV_MAX  16    /* maximum number of scatter/gather buffers */
                         /* value is not mandatory.                  */
                         /* Value must be at least 16.               */

struct  t_iovec {
```

```
    void          *iov_base;
    size_t         iov_len;
};



/*
 *  XTI LIBRARY FUNCTIONS
 */
/* XTI Library Function: t_accept - accept a connection request*/
extern int t_accept(int, int, const struct t_call *);
/* XTI Library Function: t_alloc - allocate a library structure*/
extern void *t_alloc(int, int, int);
/* XTI Library Function: t_bind - bind an address to a transport endpoint*/
extern int t_bind(int, const struct t_bind *, struct t_bind *);
/* XTI Library Function: t_close - close a transport endpoint*/
extern int t_close(int);
/* XTI Library Function: t_connect - establish a connection */
extern int t_connect(int, const struct t_call *, struct t_call *);
/* XTI Library Function: t_error - produce error message*/
extern int t_error(const char *);
/* XTI Library Function: t_free - free a library structure*/
extern int t_free(void *, int);
/* XTI Library Function: t_getinfo - get protocol-specific service */
                                                    /* information*/
extern int t_getinfo(int, struct t_info *);
/* XTI Library Function: t_getprotaddr - get protocol addresses*/
extern int t_getprotaddr(int, struct t_bind *, struct t_bind *);
/* XTI Library Function: t_getstate - get the current state*/
extern int t_getstate(int);
/* XTI Library Function: t_listen - listen for a connection indication*/
extern int t_listen(int, struct t_call *);
/* XTI Library Function: t_look - look at current event on a transport */
                                                          /* endpoint*/
extern int t_look(int);
/* XTI Library Function: t_open - establish a transport endpoint*/
extern int t_open(const char *, int, struct t_info *);
/* XTI Library Function: t_optmgmt - manage options for a transport */
                                                           /* endpoint*/
extern int t_optmgmt(int, const struct t_optmgmt *,
                           struct t_optmgmt *);
/* XTI Library Function: t_rcv - receive data or expedited data on a */
                                                         /* connection*/
extern int t_rcv(int, void *, unsigned int, int *);
/* XTI Library Function: t_rcvconnect - receive the confirmation from */
                                                 /* a connection request */
extern int t_rcvconnect(int, struct t_call *);
/* XTI Library Function: t_rcvdis - retrieve information from disconnect*/
extern int t_rcvdis(int, struct t_discon *);
/* XTI Library Function: t_rcvrel - acknowledge receipt of */
/* an orderly release indication */
extern int t_rcvrel(int);
/* XTI Library Function: t_rcvreldata - receive an orderly release */
/* indication or confirmation containing user data                  */
extern int t_rcvreldata(int, struct t_discon *)
/* XTI Library Function: t_rcvudata - receive a data unit*/
extern int t_rcvudata(int, struct t_unitdata *, int *);
/* XTI Library Function: t_rcvuderr - receive a unit data error indication*/
```

```
extern int t_rcvuderr(int, struct t_uderr *);
/* XTI Library Function: t_rcvv - receive data or expedited data sent*/
/*                              over a connection and put the data    */
/*                              into one or more noncontiguous buffers*/
extern int t_rcvv(int, struct t_iovec *, unsigned int, int *);
/* XTI Library Function: t_rcvvudata - receive a data unit into one  */
/*                                     or more noncontiguous buffers*/
extern int t_rcvvudata(int, struct t_unitdata *, struct t_iovec *, \
                                    unsigned int, int *);
/* XTI Library Function: t_snd - send data or expedited data over a */
                                                        /* connection */
extern int t_snd(int, void *, unsigned int, int);
/* XTI Library Function: t_snddis - send user-initiated disconnect request*/
extern int t_snddis(int, const struct t_call *);
/* XTI Library Function: t_sndrel - initiate an orderly release*/
extern int t_sndrel(int);
/* XTI Library Function: t_sndreldata - initiate or respond to an */
                                /* orderly release with user data */
extern int t_sndreldata(int, struct t_discon *);
/* XTI Library Function: t_sndudata - send a data unit*/
extern int t_sndudata(int, const struct t_unitdata *);
/* XTI Library Function: t_sndv - send data or expedited data,  */
/*            from one or more noncontiguous buffers, on a connection*/
extern int t_sndv(int, const struct t_iovec *, unsigned int, int);
/* XTI Library Function: t_sndvudata - send a data unit from one or  */
/*                                     more non-contiguous buffers*/
extern int t_sndvudata(int, struct t_unitdata *, struct t_iovec *, unsigned int);
/* XTI Library Function: t_strerror - generate error message string */
extern const char *t_strerror(int);
/* XTI Library Function: t_sync - synchronise transport library*/
extern int t_sync(int);
/* XTI Library Function: t_sysconf - get configurable XTI variables */
extern int t_sysconf(int);
/* XTI Library Function: t_unbind - disable a transport endpoint*/
extern int t_unbind(int);

/*
 * Protocol-specific service limits.
 */
struct t_info {
  t_scalar_t addr;     /*max size of the transport protocol address         */
  t_scalar_t options;  /*max number of bytes of protocol-specific options   */
  t_scalar_t tsdu;     /*max size of a transport service data unit          */
  t_scalar_t etsdu;    /*max size of expedited transport service data unit  */
  t_scalar_t connect;  /*max amount of data allowed on connection           */
                       /*establishment functions                           */
  t_scalar_t discon;   /*max data allowed on t_snddis, t_rcvdis,            */
                       /*t_sndreldata and t_rcvreldata functions           */
  t_scalar_t servtype; /*service type supported by transport provider       */
  t_scalar_t flags;    /*other info about the transport provider           */
};

/*
 * Service type defines.
 */
```

```
#define T_COTS       01   /* connection-mode transport service */
#define T_COTS_ORD   02   /* connection-mode with orderly release */
#define T_CLTS       03   /* connectionless-mode transport service */

/*
 * Flags defines (other info about the transport provider).
 */

#define T_SENDZERO     0x001   /* supports 0-length TSDUs */
#define T_ORDRELDATA   0x002   /* supports orderly release data */

/*
 * netbuf structure.
 */

struct netbuf {
     unsigned int   maxlen;
     unsigned int   len;
     void           *buf;
};

/*
 * t_opthdr structure
 */
struct t_opthdr {
     t_uscalar_t len;      /* total length of option; that is,  */
                           /* sizeof (struct t_opthdr) + length */
                           /* of option value in bytes          */
     t_uscalar_t level;    /* protocol affected                 */
     t_uscalar_t name;     /* option name                       */
     t_uscalar_t status;   /* status value                      */
/* implementation may add padding here */
};
/*
 * t_bind - format of the address arguments of bind.
 */

struct t_bind {
     struct netbuf  addr;
     unsigned int   qlen;
};

/*
 * Options management structure.
 */

struct t_optmgmt {
     struct netbuf  opt;
     t_scalar_t     flags;
};

/*
 * Disconnection structure.
 */
```

```
struct t_discon {
     struct netbuf   udata;      /* user data */
     int             reason;     /* reason code */
     int             sequence;   /* sequence number */
};

/*
 * Call structure.
 */

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

/*
 * Datagram structure.
 */

struct t_unitdata {
     struct netbuf   addr;    /* address */
     struct netbuf   opt;     /* options */
     struct netbuf   udata;   /* user data */
};

/*
 * Unitdata error structure.
 */

struct t_uderr {
     struct netbuf   addr;    /* address */
     struct netbuf   opt;     /* options */
     t_scalar_t      error;   /* error code */
};

/*
 * The following are structure types used when dynamically
 * allocating the above structures via t_alloc( ).
 */

#define T_BIND       1   /* struct t_bind */
#define T_OPTMGMT    2   /* struct t_optmgmt */
#define T_CALL       3   /* struct t_call */
#define T_DIS        4   /* struct t_discon */
#define T_UNITDATA   5   /* struct t_unitdata */
#define T_UDERROR    6   /* struct t_uderr */
#define T_INFO       7   /* struct t_info */

/*
 * The following bits specify which fields of the above
 * structures should be allocated by t_alloc( ).
 */
```

```
#define T_ADDR    0x01    /* address */
#define T_OPT     0x02    /* options */
#define T_UDATA   0x04    /* user data */
#define T_ALL     0xffff  /* all the above fields supported */

/*
 * The following are the states for the user.
 */

#define T_UNBND      1  /* unbound */
#define T_IDLE       2  /* idle */
#define T_OUTCON     3  /* outgoing connection pending */
#define T_INCON      4  /* incoming connection pending */
#define T_DATAXFER   5  /* data transfer */
#define T_OUTREL     6  /* outgoing release pending */
#define T_INREL      7  /* incoming release pending */

/*
 * General purpose defines.
 */

#define  T_YES            1
#define  T_NO             0
#define  T_NULL           0
#define  T_ABSREQ    0x8000
#define  T_INFINITE     (-1)
#define  T_INVALID      (-2)

/*
 *  Definitions for t_sysconf
 */
#define _SC_T_IOV_MAX     1
/*
 * General definitions for option management
 */
#define T_UNSPEC  (~0 - 2)  /* applicable to u_long, t_scalar_t, char .. */
#define T_ALLOPT  0

/*
 * The following T_OPT_FIRSTHDR, T_OPT_NEXTHDR and T_OPT_DATA macros
 * have the semantics required by the standard. They are used
 * to store and read multiple variable length objects delimited by
 * a 'header' descriptor and the variable length value content
 * while allowing aligned access to each in an arbitrary memory buffer.
 *
 * The required minimum alignment (based on types used internally
 * in the specification for header and data alignment is
 * "sizeof(t_uscalar_t)"
 *
 * The definitions shown for macro bodies are examples only and
 * other forms are not only possible but are specifically permitted.
 *
 * The examples shown assume that the implementation chooses to
 * align the header and data parts at the required minimum of
 * "sizeof(t_uscalar_t). Stricter alignment is permitted by
 * an implementation.
 *
 * Helper macros starting with "_T" prefix are used below.
```

```
 * These are not a requirement of the specification and only
 * used for constructing example macro body definitions.
 */

/*
 * Helper macro
 * _T_USCALAR_ALIGN - macro aligns to "sizeof (t_uscalar_t) boundary
 */
#define _T_USCALAR_ALIGN(p)  (((uintptr_t)(p) + (sizeof (t_scalar_t)-1))\
                                    & ~(sizeof (t_scalar_t)-1))

/*
 * struct t_opthdr *T_OPT_FIRSTHDR(struct netbuf *nbp):
 *     Get aligned start of first option header
 *
 * This implementation assumes option buffers are allocated by
 * t_alloc() and hence aligned to start any sized object
 * (including option header) is guaranteed.
 */
#define T_OPT_FIRSTHDR(nbp)    \
    ((((char *)(nbp)->buf + sizeof (struct t_opthdr)) <= \
        (char *)(nbp)->buf + (nbp)->len) ? \
        (struct t_opthdr *)(nbp)->buf : (struct t_opthdr *)0)

/*
 * unsigned char *T_OPT_DATA(struct t_opthdr *tohp):
 * Get aligned start of data part after option header
 *
 * This implementation assumes "sizeof (t_uscalar_t)" as the
 * alignment size for its option data and option header with
 * no padding in "struct t_opthdr" definition.
 */
#define T_OPT_DATA(tohp)        \
        ((unsigned char *)(tohp) + sizeof (struct t_opthdr))

/*
 * struct t_opthdr *_T_NEXTHDR(char *pbuf, unsigned int buflen,
 *                             struct t_opthdr *popt):
 *
 * Helper macro for defining T_OPT_NEXTHDR macro.
 * This implementation assumes "sizeof (t_uscalar_t)" as
 * the alignment for its option data and option header.
 * Also it assumes "struct t_opthdr" definitions contain
 * no padding.
 */
#define _T_NEXTHDR(pbuf, buflen, popt) \
        (((char *)(popt) + _T_USCALAR_ALIGN((popt)->len) + \
            sizeof (struct t_opthdr) <= \
                ((char *)(pbuf) + (buflen))) ? \
        (struct t_opthdr *)((char *)(popt) + \
            _T_USCALAR_ALIGN((popt)->len) : \
            (struct t_opthdr *)0))

/*
 * struct t_opthdr *T_OPT_NEXTHDR(struct netbuf *nbp, \
 * struct t_opthdr *tohp):
 * Skip to next option header
 * This implementation assumes "sizeof (t_uscalar_t)"
 * as the alignment size for its option data and option header.
```

```
 */
#define T_OPT_NEXTHDR(nbp, tohp)  _T_NEXTHDR((nbp)->buf, \
                                     (nbp)->len, (tohp))



/* OPTIONS ON XTI LEVEL */

/*
 *  XTI Level
 */

#define   XTI_GENERIC   0xffff

/*
 *  XTI-level Options
 */

#define   XTI_DEBUG     0x0001   /* enable debugging */
#define   XTI_LINGER    0x0080   /* linger on close if data present */
#define   XTI_RCVBUF    0x1002   /* receive buffer size */
#define   XTI_RCVLOWAT  0x1004   /* receive low-water mark */
#define   XTI_SNDBUF    0x1001   /* send buffer size */
#define   XTI_SNDLOWAT  0x1003   /* send low-water mark */

/*
 * Structure used with linger option.
 */
struct t_linger {
     t_scalar_t   l_onoff;     /* option on/off */
     t_scalar_t   l_linger;    /* linger time   */
};
```

## E.2 Example <xti_osi.h> Header File

```
/* SPECIFIC ISO OPTION AND MANAGEMENT PARAMETERS */

/*
* Definition of the ISO transport classes
*/

#define   T_CLASS0   0
#define   T_CLASS1   1
#define   T_CLASS2   2
#define   T_CLASS3   3
#define   T_CLASS4   4

/*
 * Definition of the priorities.
 */

#define   T_PRITOP    0
#define   T_PRIHIGH   1
#define   T_PRIMID    2
```

```
#define   T_PRILOW    3
#define   T_PRIDFLT   4

/*
 * Definitions of the protection levels
 */

#define   T_NOPROTECT        1
#define   T_PASSIVEPROTECT   2
#define   T_ACTIVEPROTECT    4

/*
 * rate structure.
 */
struct rate {
     t_scalar_t targetvalue;       /* target value */
     t_scalar_t minacceptvalue;    /* value of minimum acceptable quality */
};

/*
 * reqvalue structure.
 */
struct reqvalue {
     struct rate    called;    /* called rate */
     struct rate    calling;   /* calling rate */
};

/*
 * thrpt structure.
 */
struct thrpt {
     struct reqvalue    maxthrpt;    /* maximum throughput */
     struct reqvalue    avgthrpt;    /* average throughput */
};

/*
 * transdel structure
 */
struct transdel {
     struct reqvalue    maxdel;    /* maximum transit delay */
     struct reqvalue    avgdel;    /* average transit delay */
};

#define   T_ISO_TP   0x0100
```

```
/*
 *  Options for Quality of Service and Expedited Data (ISO 8072:1994)
 */

#define   T_TCO_THROUGHPUT          0x0001
#define   T_TCO_TRANSDEL            0x0002
#define   T_TCO_RESERRORRATE        0x0003
#define   T_TCO_TRANSFFAILPROB      0x0004
#define   T_TCO_ESTFAILPROB         0x0005
#define   T_TCO_RELFAILPROB         0x0006
#define   T_TCO_ESTDELAY            0x0007
#define   T_TCO_RELDELAY            0x0008
#define   T_TCO_CONNRESIL           0x0009
#define   T_TCO_PROTECTION          0x000a
#define   T_TCO_PRIORITY            0x000b
#define   T_TCO_EXPD                0x000c

#define   T_TCL_TRANSDEL            0x000d
#define   T_TCL_RESERRORRATE        T_TCO_RESERRORRATE
#define   T_TCL_PROTECTION          T_TCO_PROTECTION
#define   T_TCL_PRIORITY            T_TCO_PRIORITY

/*
 *  Management Options
 */

#define   T_TCO_LTPDU               0x0100
#define   T_TCO_ACKTIME             0x0200
#define   T_TCO_REASTIME            0x0300
#define   T_TCO_EXTFORM             0x0400
#define   T_TCO_FLOWCTRL            0x0500
#define   T_TCO_CHECKSUM            0x0600
#define   T_TCO_NETEXP              0x0700
#define   T_TCO_NETRECPTCF          0x0800
#define   T_TCO_PREFCLASS           0x0900
#define   T_TCO_ALTCLASS1           0x0a00
#define   T_TCO_ALTCLASS2           0x0b00
#define   T_TCO_ALTCLASS3           0x0c00
#define   T_TCO_ALTCLASS4           0x0d00

#define   T_TCL_CHECKSUM            T_TCO_CHECKSUM
```

## E.3 Example <xti_inet.h> Header File

```
/* INTERNET-SPECIFIC ENVIRONMENT */

/*
 *  TCP level
 */

#define   T_INET_TCP   0x6

/*
 *  TCP-level Options
 */

#define T_TCP_NODELAY   0x1 /* don't delay packets to coalesce */
#define T_TCP_MAXSEG    0x2 /* get maximum segment size        */
#define T_TCP_KEEPALIVE 0x8 /* check, if connections are alive */


/*
 * Structure used with TCP_KEEPALIVE option.
 */
struct t_kpalive {
     t_scalar_t    kp_onoff;      /* option on/off      */
     t_scalar_t    kp_timeout;    /* timeout in minutes */
};


/*
 *  UDP level
 */
#define T_INET_UDP      0x11


/*
 *  UDP-level Options
 */
#define T_UDP_CHECKSUM  T_TCO_CHECKSUM  /* checksum computation */


/*
 *  IP level
 */
#define T_INET_IP      0x0


/*
 *  IP-level Options
 */
#define T_IP_OPTIONS   0x1   /* IP per-packet options            */
#define T_IP_TOS       0x2   /* IP per-packet type of service    */
#define T_IP_TTL       0x3   /* IP per-packet time to live       */
#define T_IP_REUSEADDR 0x4   /* allow local address reuse        */
#define T_IP_DONTROUTE 0x10  /* just use interface addresses     */
#define T_IP_BROADCAST 0x20  /* permit sending of broadcast msgs */
```

```
/*
 * IP_TOS precedence levels
 */

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



/*
 * IP_TOS type of service
 */

#define   T_NOTOS             0
#define   T_LDELAY            (1 << 4)
#define   T_HITHRPT           (1 << 3)
#define   T_HIREL             (1 << 2)
#define   T_LOCOST            (1 << 1)

#define   SET_TOS(prec, tos)   ((0x7 & (prec)) << 5 | (0x1c & (tos)))
```

*Appendix F*

# *Minimum OSI Functionality*

## F.1 General

The purpose of this specification is to provide a simple API exposing a minimum set of OSI Upper Layers functionality (mOSI).

### F.1.1 Rationale for using XTI-mOSI

This appendix uses the concept of a minimal set of OSI upper layer facilities that support basic communication applications. A Basic Communication Application simply requires the ability to open and close communications with a peer and to send and receive messages with a peer.

XTI-mOSI is designed specifically for Basic Communication Applications that are in one of these categories:

- applications that are to be migrated from the Internet world (TCP or UDP) or from a NetBIOS environment to OSI
- applications accessing the OSI transport service that wish to migrate to an OSI seven-layer, conformant environment
- applications that require a simple octet-stream connection between peer processes. The benefit of XTI-mOSI to these applications is that it extends the family of *transport services* that are available via a single, protocol independent, API.

### F.1.2 Migrant Applications

For the first kind of applications (those migrating to OSI or intended to work over a variety of *transport* mechanisms), the migration effort will be greatly simplified if they were already using XTI — mOSI offers several new options, but, as described later in this section, default values are generally provided.

In addition to applications already using XTI, the X Window System (X) and Internet Protocol Suite applications (in general) are examples of potential Migrant applications.

### F.1.3 OSI Functionality

mOSI is suited to applications that require only the Minimal Upper Layer facilities which are described in the profile ISO/IEC DISP 11188 — Common Upper Layer Requirements, Part 3: Minimal OSI upper layer facilities, expected to reach International Standardized Profile (ISP) status in the first half of 1995. These are:

- ACSE Kernel functional unit
- Presentation Kernel functional unit
- Session Kernel and Full Duplex functional units.

The XTI-mOSI interface provides access to OSI ACSE and Presentation services. With mOSI, the optional parameters available to the application have been selected with the intent of facilitating interoperability and diagnostic of problems. They are described later in this section.

Most applications only need the Kernel functionality. This is even true for most of the OSI standard applications: Remote Database Access (RDA), Directory (X.500), FTAM without recovery, OSI Distributed Transaction Processing (TP) without 2-phase commitment, OSI Management.

### F.1.4 mOSI API versus XAP

X/Open has developed the XAP interface (ACSE/Presentation API) to provide access to ACSE and Presentation functionality. It provides an interface which spans from implementations of minimal OSI through to full ACSE/Presentation/Session with all functional units.

XTI-mOSI has been developed to support migrant transport applications, and also applications which require a simple octet stream connection between peers.

Application designers should consider which API is most appropriate to their needs.

Applications which are being ported from other transport providers, or require a simple octet string connection, should use XTI-mOSI.

Applications which require session function units outside kernel, or which require multiple presentation contexts, or which may require additional session/presentation by mOSI in the future, should use XAP.

When only minimal OSI support is required, the user should consider the availability of XTI-mOSI and XAP on target hardware platforms when selecting an interface.

### F.1.5 Upper Layers Functionality Exposed via mOSI

These are presented as they are exposed via mOSI options and specific parameters.

#### *F.1.5.1 Naming and Addressing Information used by mOSI*

The **addr** structure (used in *t_bind*(), *t_connect*(), *t_accept*()) is a combined naming and addressing data type, identifying one end or the other of the association.

The address part is a Presentation Address. The *calling* and *called* addresses are required parameters, while the use of a *responding* address is optional.

The name part (Application Process (AP) Title, Application Entity (AE) Qualifier and the AP and AE invocation-identifiers) is always optional.

ISO Directory facilities, when available, can relate the name parts (identifying specific applications) to the addresses of the real locations where they can be accessed.

The general format of the **addr** structure can be found in Section F.5 on page 347, while its precise structure is implementation dependent.

#### *F.1.5.2 XTI Options Specific to mOSI*

- Application Context Name

  An application context name identifies a set of tasks to be performed by an application. It is exchanged during association establishment with the purpose of conveying a common understanding of the work to be done.

  This parameter is exposed to offer some negotiation capabilities to the application and to increase the chances of interoperability.

  When receiving a non suitable or unknown value from a peer application, the application may propose an alternate value or decide to terminate prematurely the association.

A default value (in the form of an Object Identifier) is provided, identifying a generic XTI-mOSI application. Its value can be found in Section F.5 on page 347.

- Presentation Contexts

  A presentation context is the association of an abstract syntax with a transfer syntax. The presentation context is used by the application to identify how the data is structured and by the OSI Application Layer to identify how the data should be encoded/decoded.

  A *generic* presentation context is defined for a stream-oriented, unstructured, data transfer service with *null* encoding:

  *abstract syntax:* The single data type of this abstract syntax is a sequence of octets that are defined in the application protocol specification as being consecutive octets on a stream-oriented transport mechanism, without regard for any semantic or other boundaries.

  *transfer syntax:* The data value shall be represented as an octet-aligned presentation data value. If two or more data values are concatenated together they are considered to be a single (longer) data value. (This is the *null* encoding rule).

  The value of the Object Identifiers for this *generic* presentation context can be found in Section F.5 on page 347.

- Presentation Context Definition and Result List, Defined Context Set

  As negotiation occurs between the peer OSI Application layers, the presentation context(s) proposed by the application may not be accepted.

  The Presentation Context Definition and Result List indicates, for each of the proposed presentation context, if it is accepted or, if not, provides a reason code; the application may choose to terminate the association prematurely if it does not suit its requirements.

## F.2 Options

Options are formatted according to the structure **t_opthdr** as described in Chapter 13 on page 149. An OSI provider compliant to this specification supports all, none or a subset of the options defined in Section F.2.1. An implementation may restrict the use of any of the options by offering them in privileged or read_only mode.

An explanation of when an application may benefit from using the XTI options specific to mOSI can be found in Section F.1 on page 331.

### F.2.1 ACSE/Presentation Connection-mode Service

The protocol level for all subsequent options is T_ISO_APCO.

All options have end-to-end significance (see Chapter 13 on page 149. They may be negotiated in the XTI states T_IDLE and T_INCON, and are read-only in all other states except T_UNINIT. The structures referenced are specified in Section F.5 on page 347.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_AP_CNTX_NAME | Object identifier item (see Section F.5 on page 347) | see text default: see text | Application Context Name |
| T_AP_PCL | Presentation Context Definition and Result list (see Section F.5 on page 347) | see text default: see text | Presentation Context Definition and Result List |

**Table F-1** APCO-level Options

**Further Remarks**

- Application Context Name

  A default value (for a *generic* XTI-mOSI application) is provided. It is defined in Section F.5 on page 347.

  The application may choose to propose, through this option, a value different from the default one. The application may also use this option to check the value returned by the peer application and decide if the association should be kept or terminated.

- Presentation Context Definition and Result List

  A default is provided: a list with one presentation context (the stream oriented, unstructured, data transfer service with *null* encoding — this is described in section Section F.1 on page 331. The abstract syntax is the default abstract syntax and the transfer syntax is the default transfer syntax, as specified in Section F.5 on page 347.

  The codes for the result of negotiation and reason for rejection are defined in Section F.5 on page 347. The responding application, after reading this option, may choose to continue or terminate the association.

  Only a single abstract syntax and transfer syntax can be used by XTI-mOSI. On *t_accept*( ), this is assumed to be the first usable abstract syntax and the first transfer syntax for that abstract syntax.

**Negotiation of Abstract and Transfer Syntax**

When initiating a connection, the application proposes one or more presentation contexts, each comprising an abstract syntax and one or more transfer syntaxes in the Presentation Context Definition and Result List option (or omits this option to select the CULR-3 defaults), and issues a *t_connect*( ).

If the connection is accepted, the Presentation Context Definition and Result List is updated to reflect the results of negotiation for each element of the context list, and a single presentation context is selected.

**Note:** If the responder accepts multiple presentation contexts, the XTI-mOSI provider aborts the connection on receipt of the A-ASSOCIATE confirm.

When responding to a remote connect, the application can specifically mark presentation contexts as rejected using the *res* field, and can re-order the syntax array to select a single transfer syntax.

On calling *t_accept*( ), the first presentation context marked as accepted is selected, and all other contexts omitted or not marked rejected-user are marked as by the provider as rejected (T_PCL_PREJ_LMT_DCS_EXCEED). In an accepted context, the provider will accept the first (or only remaining) transfer syntax.

**Note:** On return to the application from *t_listen*( ), all supportable presentation contexts are marked as accepted in the T_AP_PCL option, and all unsupportable contexts are marked as *rejected-provider*. This permits the application to return the same option value on *t_accept*( ) (or leave it unchanged) to select the first available abstract syntax and transfer syntax.

**Management Options**

No management options are defined.

### F.2.2 ACSE/Presentation Connectionless-mode Service

The protocol level for all subsequent options is T_ISO_APCL.

All options have end-to-end significance (see Chapter 13 on page 149). They may be negotiated in all XTI states except T_UNINIT. The structures referenced are specified in Section F.5 on page 347.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_AP_CNTX_NAME | Object identifier item (see Section F.5 on page 347) | see text<br>default: see text | Application Context Name |
| T_AP_PCL | Presentation Context Definition and Result list (see Section F.5 on page 347) | see text<br>default: see text | Presentation Context Definition and Result List |

**Table F-2** APCL-level Options

**Further Remarks**

- Application Context Name

  A default value (for a *generic* XTI-mOSI application) is provided. It is defined in Section F.5 on page 347.

  The application may choose to propose, through this option, a value different from the default one. The application may also use this option to check the value returned by the peer application and decide if the datagram should be kept or discarded.

- Presentation Context Definition and Result List

  In connectionless mode, the transfer syntaxes are not negotiated. Their use is determined by the sending application entity, and must be acceptable by the receiving application entity. A default value is provided by XTI: a list with one element, the *generic* presentation context (the stream-oriented, unstructured, data transfer service with *null* encoding described in Section F.1 on page 331). The corresponding abstract and transfer syntaxes are specified in Section F.5 on page 347.

  Only a single abstract syntax and transfer syntax can be used by connectionless-mode XTI-mOSI. If more than one presentation context is present in the options list for *t_sndudata*( ), the first is used.

**Management Options**

No management options are defined.

### F.2.3 Transport Service Options

Some of the options defined for XTI ISO Transport Connection-mode Service or Transport Connectionless-mode Service may be made available to mOSI users: the Options for Quality of Service.

These Options are defined in Section A.2.1.1 on page 266 and Section A.2.2.1 on page 270. The Quality of Service parameters are passed directly by the OSI Upper Layers to the Transport Layer. These options can thus be used to specify OSI Upper Layers quality of service parameters via XTI.

This facility is implementation dependent. An attempt to specify an unsupported option will return with the status field set to T_NOTSUPPORT.

None of these options are available with an ISO-over-TCP transport provider.

## F.3 Functions

Functions *rcvreldata*( ) (see *t_rcvreldata*( ) on page 209) and *sndreldata*( ) (see *t_sndreldata*( ) on page 225) were introduced as part of this XTI-mOSI functionality. The rationale for this is that for ISO ACSE providing an orderly release mechanism, user data is a parameter of the release service, so when mapping XTI primitives to ACSE/Presentation (XTI-mOSI), disconnection user data may be received from peer applications. Although abortive release primitives (t_snddis, t_rcvdis) permit sending and receiving of user data, orderly release primitives (t_sndrel, t_rcvrel) do not. Therefore, new functions having a user data parameter *t_rcvreldata*( ) and *t_sndreldata*( ) were were added to provide the necessary support to handle this user data.

*t_accept*( ) If *fd* is not equal to *resfd*, *resfd* should either be in state T_UNBND or be in state T_IDLE with the *qlen* parameter set to 0.

The addr parameter passed to/returned from *t_bind*( ) when resfd is bound may be different from the addr parameter corresponding to fd.

The opt parameter may be used to change the Application Context Name received.

*t_alloc*( ) No special considerations for mOSI providers.

*t_bind*( ) The addr field of the **t_bind** structure represents the local presentation address and optionally the local AP Title, AE Qualifier, AP and AE invocation-identifiers (see Section F.1 on page 331 and Section F.5 on page 347 for more details).

This local addr 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 (fd, resfd), bound to different addresses, are used.

*t_close*( ) 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 disconnection reason will be T_AC_ABRT_NSPEC.

*t_connect*( ) The *sndcall→addr* structure specifies the Called Presentation Address. The *rcvcall→addr* structure specifies the Responding Presentation Address. The structure may also be used to assign values for the Called AP Title, Called AE Qualifier, Called AP invocation-identifier and Called AE invocation-identifier.

Before the call, the *sndcall→opt* structure may be used to request an Application Context name or Presentation Context different from the default value.

*t_error*( ) No special considerations for mOSI providers.

*t_free*( ) No special considerations for mOSI providers.

*t_getinfo*( ) The information supported by *t_getinfo*( ) reflects the characteristics of the *transport* connection, or if no connection is established, the default characteristics of the underlying OSI layers. In all possible states except T_DATAXFER, the function *t_getinfo*( ) returns in the parameter info the same information as was returned by *t_open*( ). In state T_DATAXFER, however, the information returned in *info→connect* and *info→discon* may differ.

The parameters of the *t_getinfo*( ) function are summarised in the table below.

| Parameters | Before call | After call | |
|---|---|---|---|
| | | **Connection-mode** | **Connectionless-mode** |
| fd | x | / | / |
| info→addr | / | x | x |
| info→options | / | x | x |
| info→tsdu | / | T_INFINITE (–1) | T_INFINITE (–1) |
| info→etsdu | / | (T_INVALID (–2) | T_INVALID (–2) |
| info→connect | / | x | T_INVALID (–2) |
| info→discon | / | x | T_INVALID (–2) |
| info→servtype | / | T_COTS_ORD | T_CLTS |
| info→flags | / | 0 | 0 |

x equals an integral number greater than 0.

The values of the parameters in the **t_info** structure for the *t_getinfo*( ) function reflect the mOSI provider particularities.

- connect, discon

  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.

- flags

  mOSI does not support sending of TSDU of zero length, so this value equals 0.

*t_getprotaddr*( ) The protocol addresses are naming and addressing parameters as defined in Section F.1 on page 331 and Section F.5 on page 347.

*t_getstate*( ) No special considerations for mOSI providers.

*t_listen*( ) The *call→addr* structure contains the remote Calling Presentation Address and the remote Calling AP Title, AE Qualifier, and AP and AE invocation identifiers if received.

Incoming user data encoded as multiple presentation data values will cause the TBADDATA error to be returned.

*t_look*( ) Since expedited data is not supported for a mOSI provider, T_EXDATA and T_GOEXDATA events cannot occur.

*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.

The parameters of the *t_open*( ) function are summarised in the table below.

| Parameters | Before call | After call | |
|---|---|---|---|
| | | Connection-mode | Connectionless-mode |
| name | x | / | / |
| oflag | | / | / |
| info→addr | / | x | x |
| info→options | / | x | x |
| info→tsdu | / | T_INFINITE (–1) | T_INFINITE (–1) |
| info→etsdu | / | T_INVALID (–2) | T_INVALID (–2) |
| info→connect | / | x | T_INVALID (–2) |
| info→discon | / | x | T_INVALID (–2) |
| info→servtype | / | T_COTS_ORD | T_CLTS |
| info→flags | / | 0 | 0 |

x equals an integral number greater than 0.

The values of the parameters in the **t_info** structure reflect mOSI limitations as follows:

- connect, discon

  These values are limited by the version of the session supported by the mOSI provider, and are generally much larger than those supported by an ISO Transport or TCP provider.

- flags

  mOSI does not support sending of *TSDU* of zero length, so this value equals 0.

**Note:** The name (device file) parameter passed to *t_open*( ) will differ when the application accesses an mOSI provider or an ISO Transport provider.

*t_optmgt*( ) The options available with mOSI providers are described in section Section F.2 on page 334.

*t_rcv*( ) The flags parameter will never be set to [T_EXPEDITED], as expedited data transfer is not supported.

*t_rcvconnect*( ) The *call→addr* structure specifies the remote Responding Presentation Address, and the remote responding AP Title, AE Qualifier, and AP and AE invocation identifiers if received.

The *call→opt* structure may also contain an Application Context Name and/or Presentation Context Definition Result List.

*t_rcvdis*( ) Possible values for disconnection reason codes are specified in Section F.5 on page 347.

*t_rcvrel*( ) With this primitive, user data cannot be received on normal release: any user data in the received flow is discarded (see *t_rcvreldata*( ) on page 209).

*t_rcvudata*( ) The *unitdata→addr* structure specifies the remote Presentation address, and optionally the remote AP Title, AE Qualifier, AP and AE invocation-identifiers. If the T_MORE flag is set, an additional *t_rcvudata*( ) call is needed to retrieve the entire A-UNIT-DATA service unit. Only normal data is returned via the *t_rcvudata*( ) call.

| | |
|---|---|
| *t_rcvuderr*( ) | This function is not supported by a mOSI provider since badly formed A-UNIT-DATA APDUs are discarded. |
| *t_snd*( ) | Zero-length TSDUs are not supported.<br>Since expedited data transfer is not supported for a mOSI provider, the parameter flags shall not have [T_EXPEDITED] set. |
| *t_snddis*( ) | No special considerations for mOSI providers. |
| *t_sndrel*( ) | With this primitive, user data cannot be sent on normal release (see *t_sndreldata*( ) on page 225). |
| *t_sndudata*( ) | The *unitdata→addr* structure specifies the remote Presentation address, and optionally the remote AP Title, AE Qualifier, AP and AE invocation-identifiers. Only normal data is sent via the *t_sndudata*( ) call. |
| *t_strerror*( ) | No special considerations for mOSI providers. |
| *t_sync*( ) | No special considerations for mOSI providers. |
| *t_unbind*( ) | No special considerations for mOSI providers. |

## F.4 Implementors´ Notes

### F.4.1 Upper Layers FUs, Versions and Protocol Mechanisms

The implementation negotiates:

Session: Kernel, Full Duplex, version 2, or version 1 if version 2 not supported, no segmentation.

Other session protocol mechanisms are out of scope, except Basic Concatenation which is mandatory and transparent to the application.

Presentation: Kernel, Normal Mode

ACSE: Kernel

If invalid (non-negotiable) options are requested by the peer and detected by the provider once the association is already established (such as the ACSE presentation context missing in the Defined Context Set), the association is rejected via an A-(P)-ABORT generated by the implementation.

### F.4.2 Mandatory and Optional Parameters

- If the Local Presentation Address is not passed to *t_bind*( ) in *req→addr*, then it is returned in *ret→addr*.
- The remote (called) Presentation Address (in *t_connect*( ), *sndcall→addr*) parameter must be explicitly set by the application.
- The following parameters are mandatory for the protocol machine, but default values are provided. If the application does not wish to set the corresponding parameter, the default value will be used. The default value may be changed through *t_optmgt* (see Section F.2 on page 334):
  - — Application Context Name (opt parameter)
  - — Presentation Context List (opt parameter).

    The presentation context of ACSE is required and used. The user should not request it as the implementation will insert it automatically in the context list.

    If the user does not specifically request an Application Context name via the opt parameter of *t_accept*( ) (that is, for the A-Associate response), the implementation uses the Application Context name that was received in the A-Associate indication.
- The following parameters are optional for the protocol and default values of null are defined. If the application does not set them otherwise, they are omitted from the outgoing protocol stream.
  - — local AP-title (in *t_bind*( ), *req→addr*)
  - — called AP-title (in *t_connect*( ), *sndcall→addr*)
  - — responding AP-title (if *t_accept*( ) specifies a new accepting endpoint *resfd*, in the protocol address bound to *resfd*)
  - — local AE-qualifier (in *t_bind)*( ), *req→addr*)
  - — called AE-qualifier (in *t_connect*( ), *sndcall→addr*)
  - — responding AE-qualifier (if *t_accept*( ) specifies a new accepting endpoint *resfd*, in the protocol address bound to *resfd*).

— local AP and AE invocation-identifiers (in *t_bind*( ), *req→addr*)

— called AP and AE invocation-identifiers (in *t_connect*( ), *sndcall→addr*)

— responding AP and AE invocation-identifiers (if *t_accept*( ) specifies a new accepting endpoint *resfd*, in the protocol address bound to *resfd*).

- The following parameters are optional for the protocol machine and not supported through the XTI interface. Their handling is implementation-defined. Received values in the incoming protocol stream, if any, are discarded:

  — ACSE Protocol Version (default= version 1)

  — Presentation Protocol Version (default= version 1)

  — ACSE Implementation Information

  — Session connection identifiers.

  During association establishment (that is, before the XTI-mOSI provider negotiates acceptance of a single abstract syntax/transfer syntax pair), an XTI-mOSI application initiating the association will only send a single presentation data value in the user information parameter. The XTI-mOSI provider will insure that the first abstract syntax and transfer syntax pair being negotiated is the one required for its encoding.

### F.4.3 Mapping XTI Functions to ACSE/Presentation Services

In the following tables, the definition of which parameters are mandatory and which are optional can be found in ISO/IEC DISP 11183 — Common Upper Layers Requirements, part 3 (see reference **CULR**).

#### *F.4.3.1 Connection-mode Services*

**Association Establishment (successful, unsuccessful)**

**Note:** XTI does not support the concept of a negative association establishment; that is, the equivalent of a negative A-ASSOCIATE response. That is, an XTI-mOSI implementation does not generate an AARE- APDU.

To reject an association request, the responding application issues *t_snddis*( ), which is mapped to a A-ABORT.

However, a negative A-ASSOCIATE confirm (AARE- APDU) may be received from a non-XTI OSI peer. The negative A-ASSOCIATE confirm event is mapped to *t_rcvdis( )*.

**Table F-3** Association Establishment

| XTI call | Parameter | Service | Parameter |
|---|---|---|---|
| t_connect | | A-ASSOCIATE req | |
| | sndcall→addr | | Called Presentation Address |
| | sndcall→addr (1) | | Called AP Title |
| | sndcall→addr (1) | | Called AE Qualifier |
| | sndcall→addr | | Called AP invocation-identifier |
| | sndcall→addr | | Called AE invocation-identifier |
| | sndcall→opt (2) | | Application Context Name |
| | sndcall→opt (3) | | P-context Definition and Result List |
| | sndcall→udata | | User Information |
| {t_bind} | req \| ret→addr | | Calling Presentation Address |
| {t_bind} | req \| ret→addr | | Calling AP Title |
| {t_bind} | req \| ret→addr | | Calling AE Qualifier |
| t_listen | | A-ASSOCIATE ind | |
| | call→addr | | Calling Presentation Address |
| | call→addr (1) | | Calling AP Title |
| | call→addr (1) | | Calling AE Qualifier |
| | call→opt | | Application Context Name |
| | call→opt (4) | | P-context Definition and Result List |
| | call→udata | | User Information |
| {t_bind} | req \| ret→addr | | Called Presentation Address |
| {t_bind} | req \| ret→addr (1) | | Called AP Title |
| {t_bind} | req \| ret→addr (1) | | Called AE Qualifier |
| {t_bind} | req \| ret→addr | | Calling AP invocation-identifier |
| {t_bind} | req \| ret→addr | | Calling AE invocation-identifier |
| t_accept | | A-ASSOCIATE rsp+ | |
| | call→addr | | not used: Calling Presentation Address |
| | call→opt | | Application Context Name |
| | call→opt | | P-context Definition and Result List |
| | call→udata | | User Information |
| {internal} | ::="accepted" | | Result |
| {t_bind} | req \| ret→addr | | Responding Presentation Address |
| {t_bind} | req \| ret→addr (1) | | Responding AP Title |
| {t_bind} | req \| ret→addr (1) | | Responding AE Qualifier |
| {t_bind} | req \| ret→addr | | Responding AP invocation-identifier |
| {t_bind} | req \| ret→addr | | Responding AE invocation-identifier |
| not sent | | A-ASSOCIATE rsp- | |
| t_connect | (synchronous mode) | A-ASSOCIATE cnf+ | |
| | rcvcall→addr | | Responding Presentation Address |
| | rcvcall→addr | | Responding AP Title |
| | rcvcall→addr | | Responding AE Qualifier |
| | rcvcall→addr | | Responding AP invocation-identifier |
| | rcvcall→addr | | Responding AE invocation-identifier |
| | rcvcall→opt | | Application Context Name |
| | rcvcall→opt | | P-context Definition and Result List |
| | rcvcall→udata | | User Information |
| {internal} | ::="accepted" | | Result |
| {internal} | ::="ACSE service-user" | | Result Source |
| t_rcvconnect | (asynchronous mode) | A-ASSOCIATE cnf+ | |
| | call→addr | | Responding Presentation Address |
| | call→addr | | Responding AP Title |
| | call→addr | | Responding AE Qualifier |

| XTI call | Parameter | Service | Parameter |
|---|---|---|---|
| | call→addr | | Responding AP invocation-identifier |
| | call→addr | | Responding AE invocation-identifier |
| | call→opt | | Application Context Name |
| | call→opt | | P-context Definition and Result List |
| | call→udata | | User Information |
| {discarded} | ::="accepted" | | Result |
| {discarded} | ::="ACSE service-user" | | Result Source-diagnostic |
| | | | |
| t_rcvdis | | A-ASSOCIATE cnf- | |
| | discon→udata | | User Information |
| | discon→reason (5) | | Result |
| {internal} | ACSE serv-user \| pres serv-prov | | Result Source-diagnostic |
| | {discarded} | | Application Context Name |
| | {discarded} | | P-context Definition and Result List |

**Notes:**

(1) if either the AP title or AE qualifier is selected for sending, the other must be selected.

(2) *sndcall→opt* or, if no option specified, default value

(3) *sndcall→opt* or, if no option specified, default value, with ACSE added by provider

(4) *call→opt* with ACSE context removed from the list passed to user

(5) combines Result and Result Source-diagnostic

**Data Transfer**

| XTI call | Parameter | Service | Parameter |
|---|---|---|---|
| t_snd | | P-DATA req | |
| | buf | | User Data |
| | | | |
| t_rcv | | P-DATA ind | |
| | buf | | User Data |

**Table F-4** Data Transfer

**Association Release (orderly, abortive)**

This table makes the assumption that the XTI-mOSI provider supports the orderly release facility with user data ( *t_sndreldata*( ) — see *t_rcvreldata*( ) on page 209, and *t_rcvreldata*( ) — see *t_rcvreldata*( ) on page 209). When this is not the case, User Information is not sent, Reason is supplied via an internal mechanism with A-RELEASE request and response, User Information and Reason received in A-RELEASE indication and confirmation are discarded.

| XTI call | Parameter | Service | Parameter |
|---|---|---|---|
| t_sndrel2 | | A-RELEASE req | |
| | reldata→reason | | Reason |
| | reldata→udata | | User Information |
| t_rcvreldata | | A-RELEASE ind | |
| | reldata→reason | | Reason |
| | reldata→udata | | User Information |
| t_sndrel2 | | A-RELEASE rsp | |
| | reldata→reason | | Reason |
| | reldata→udata | | User Information |
| t_rcvreldata | | A-RELEASE cnf | |
| | reldata→reason | | Reason |
| | reldata→udata | | User Information |
| t_snddis | | A-ABORT req | |
| | n/s | | Diagnostic |
| | call→udata | | User Information |
| t_rcvdis | | A-ABORT ind | |
| | discon→reason | | Diagnostic |
| | discon→udata | | User Information |
| t_rcvdis | | A-P-ABORT ind | |
| | discon→reason | | Diagnostic |

**Table F-5** Association Release

### *F.4.3.2 Connectionless-mode Services*

| XTI call | Parameter | Service | Parameter |
|---|---|---|---|
| t_sndudata | | A-UNIT-DATA source | |
| | unitdata→addr | | Called Presentation Address |
| | unitdata→addr | | Called AP Title |
| | unitdata→addr | | Called AE Qualifier |
| | unitdata→addr | | Called AP invocation-identifier |
| | unitdata→addr | | Called AE invocation-identifier |
| | unitdata→opt (1) | | Application Context Name |
| | unitdata→opt (2) | | P-context Definition and Result List |
| | unitdata→udata | | User Information |
| {t_bind} | req \| ret→addr | | Calling Presentation Address |
| {t_bind} | req \| ret→addr | | Calling AP Title |
| {t_bind} | req \| ret→addr | | Calling AE Qualifier |
| {t_bind} | req \| ret→addr | | Calling AP invocation-identifier |
| {t_bind} | req \| ret→addr | | Calling AE invocation-identifier |
| t_rcvudata | | A-UNIT-DATA sink | |
| | unitdata→addr | | Calling Presentation Address |
| | unitdata→addr | | Calling AP Title |
| | unitdata→addr | | Calling AE Qualifier |
| | unitdata→addr | | Calling AP invocation-identifier |
| | unitdata→addr | | Calling AE invocation-identifier |
| | unitdata→opt | | Application Context Name |
| | unitdata→opt (3) | | P-context Definition and Result List |
| | unitdata→udata | | User Information |
| {t_bind} | req \| ret→addr | | Called Presentation Address |
| {t_bind} | req \| ret→addr | | Called AP Title |
| {t_bind} | req \| ret→addr | | Called AE Qualifier |
| {t_bind} | req \| ret→addr | | Called AP invocation-identifier |
| {t_bind} | req \| ret→addr | | Called AE invocation-identifier |

**Table F-6** Connectionless-mode ACSE Service

**Notes:**

(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

## F.5 Option Data Types and Structures

### SPECIFIC ISO ACSE/PRESENTATION OPTIONS

#### Naming and Addressing Datatype

The buf[ ] part of the *addr* structure is an mosiaddr structure defined in the following way:

```
struct t_mosiaddr {
    t_uscalar_t     flags;
    t_scalar_t      osi_ap_inv_id;
    t_scalar_t      osi_ae_inv_id;
    unsigned int    osi_apt_len;
    unsigned int    osi_aeq_len;
    unsigned int    osi_paddr_len;
    unsigned char   osi_addr[T_AP_MAX_ADDR];
};
```

where:

- the *flags* field indicates the validity of the contents of the invocation identifier fields within the structure. One or more of the following bits may be set:

| | |
|---|---|
| T_OSI_AP_IID_BIT | The contents of the osi_ap_inv_id field is valid |
| T_OSI_AE_IID_BIT | The contents of the osi_ae_inv_id field is valid |

  Unused bit in *flags* must be set zero by the user when creating a **t_mosiaddr** structure for sending, and should be ignore by the user on receipt.

  In a **t_mosiaddr** structure, a bit set in *flags* indicates the presence of the corresponding invocation identifier in the PDU. Similarly a bit not set indicates absence of the corresponding invocation identifier in the PDU.

- the AP Title starts at
  `osi_addr[0]`
- the AE Qualifier starts at
  `osi_addr[T_ALIGN(osi_apt_len)]`
- the Presentation Address is at
  `osi_addr[T_ALIGN(osi_apt_len)+T_ALIGN(osi_aeq_len)]`
- T_AP_MAX_ADDR is an implementation-defined constant.

The application is responsible for encoding/decoding the AP title and AE qualifier; alternatively, a lookup routine may be provided (outside the scope of this specification).

#### ACSE/Presentation Option Levels and Names

```
#define T_ISO_APCO      0x0200
#define T_ISO_APCL      0x0300
#define T_AP_CNTX_NAME  0x1
#define T_AP_PCL        0x2
```

### Object Identifier Representation within Options

The application context, abstract syntax and transfer syntax all utilise object identifiers. An object identifier is held in the BER encoded form as a variable length item, whose length can be inferred from the length of the option, as in the following macro:

```
#define T_OPT_VALEN(opt) (opt→len - sizeof(struct t_opthdr)).
```

The application is responsible for encoding/decoding the object identifier value. Alternatively, a lookup routine may be provided (outside the scope of this specification).

### Application Context Name Option

The application context name option consists of an object identifier item as defined above.

### Presentation Context Definition and Result List Option

The Presentation Context Definition and Result List option is used to propose a presentation context, giving its abstract and transfer syntax, and to hold the result of negotiation of a presentation context.

The presentation context definition and result list option is a variable sized option consisting of a `t_scalar_t` giving the number of presentation contexts followed by an array of that number of presentation context item offset elements. Each element is defined as:

```
struct t_ap_pco_el  {
    t_scalar_t    count;
    t_scalar_t    offset;
}
```

Each presentation context items offset element gives: the *count* of syntax entries in the presentation context item (including the abstract syntax entry) and the *offset* of the presentation context item from the beginning of the Presentation Context Definition and Result List option value.

Each presentation context item consists of a header followed by an array.

The header is defined as:

```
struct t_ap_pc_item  {
    t_scalar_t     pci;
    t_scalar_t     res;
}
```

where *pci* is a unique odd integer (if this is zero in the *sndcall* argument of *t_connect*( ), the provider will substitute an appropriate value), and *res* is the result of negotiation. The array of syntax offset elements immediately follows the header. Each element is defined as:

```
struct t_ap_syn_off  {
    t_scalar_t    size;
    t_scalar_t    offset;
}
```

where *size* is the length of the syntax object identifier contents, and *offset* is the offset of the object identifier for the syntax from the beginning of the Presentation Context Definition and Result List option value.

The first element in the array of syntax offset elements refers to the abstract syntax the second to the first transfer syntax, and so on.

The following values are used in the *res* field of a presentation context item:

```
#define  T_PCL_ACCEPT                0x0000
                    /*pres. context accepted                      */
#define  T_PCL_USER_REJ              0x0100
                    /*pres. context rejected by peer application */
#define  T_PCL_PREJ_RSN_NSPEC        0x0200
                    /*prov. reject: no reason specified          */
#define  T_PCL_PREJ_A_SYTX_NSUP      0x0201
                    /*prov. reject: abstract syntax not supported*/
#define  T_PCL_PREJ_T_SYTX_NSUP      0x0202
                    /*prov. reject:transfer syntax not supported */
#define  T_PCL_PREJ_LMT_DCS_EXCEED   0x0203
                    /*prov. reject: local limit on DCS exceeded  */
```

For the default abstract syntax, transfer syntax and application context, this Appendix uses object identifiers which are specified in the profile (ISO/IEC pDISP 11188 - Common Upper Layer Requirements (CULR), Part 3: Minimal OSI upper layer facilities - OIW/EWOS working document). Thus the descriptions provided in this Appendix are informative only.

**Default Abstract Syntax for mOSI**

The following OBJECT IDENTIFIER have been defined in CULR part 3:

{iso(1) standard(0) curl(11188) mosi(3) default-abstract-syntax(1) version(1)}

This object identifier can be used as the abstract syntax when the application protocol (above ACSE) can be treated as single presentation data values (PDVs). Each PDV is a sequence of consecutive octets without regard for semantic or other boundaries. The object identifier may also be used when, for pragmatic reasons, the actual abstract syntax of the application is not identified in Presentation Layer negotiation.

**Notes:**

1. Applications specified using ASN.1 should not use the default abstract syntax.
2. As this object identifier is used by all applications using the default abstract syntax for mOSI, it cannot be used to differentiate between applications. One of the ACSE parameters; for example, AE Title or Presentation address, may be used to differentiate between applications.

**Default Transfer Syntax for mOSI**

If the default transfer syntax and the abstract syntax are identical, the OBJECT IDENTIFIER for the default abstract syntax is used. If they are not identical, the OBJECT identifier for the default transfer syntax is:

{iso(1) standard(0) curl(11188) mosi(3) default-transfer-syntax(2) version(1)}

**Note:** In the presentation data value of the PDV list of Presentation Protocol or in the encoding of User Information of ACSE Protocol, only *octet-aligned* or *arbitrary* can be used for default transfer syntax for mOSI. *Single-ASN1-type* cannot be used for default transfer syntax for mOSI.

**Default Application Context for mOSI**

The following OBJECT IDENTIFIER has been defined in Common Upper Layer Requirements (CULR) part 3:

{iso(1) standard(0) curl(11188) mosi(3) default-application-context(3) version(1)}

This application context supports the execution of any application using the default abstract syntax for mOSI.

**Reason Codes for Disconnections**

```
#define T_AC_U_AARE__NONE  0x0001     /* connection rejected by      */
                                      /* peer user: no reason given  */
#define T_AC_U_AARE_ACN    0x0002     /* connection rejected:        */
                                      /* application context name    */
                                      /* not supported               */
#define T_AC_U_AARE_APT    0x0003     /* connection rejected:        */
                                      /* AP title not recognised     */
#define T_AC_U_AARE_AEQ    0x0005     /* connection rejected:        */
                                      /* AE qualifier not recognised */
#define T_AC_U_AARE_PEER_AUTH 0x000e  /* connection rejected:        */
                                      /* authentication required     */
#define T_AC_P_ABRT_NSPEC  0x0011     /* aborted by peer provider:   */
                                      /* no reason given             */
#define T_AC_P_AARE_VERSION 0x0012    /* connection rejected:        */
                                      /* no common version           */
```

Other reason codes may be specified as implementation defined constants. In order to be portable, an application should not interpret such information, which should only be used for troubleshooting purposes.

## F.6 <xti_mosi.h> Header File

This section presents the additional header file information for XTI-mOSI.

Implementations supporting XTI-mOSI will provide equivalent definitions in <xti_mosi.h>. XTI-mOSI programs should include <xti_mosi.h> as well as <xti.h>.

```
/* mosi address structure */

struct t_mosiaddr {
    t_uscalar_t     flags;
    t_scalar_t      osi_ap_inv_id;
    t_scalar_t      osi_ae_inv_id;
    unsigned int    osi_apt_len;
    unsigned int    osi_aeq_len;
    unsigned int    osi_paddr_len;
    unsigned char   osi_addr[MAX_ADDR];
};

#define T_ISO_APCO      0x0200
#define T_ISO_APCL      0x0300
#define T_AP_CNTX_NAME  0x1
#define T_AP_PCL        0x2

#define T_OPT_VALEN(opt) (opt→len - sizeof(struct t_opthdr)).

/* presentation context definition and result list option */

struct t_ap_pco_el {
    t_scalar_t count;
    t_scalar_t offset;
}

/* presentation context item header */

struct t_ap_pc_item {
    t_scalar_t pci;      /* unique odd integer    */
    t_scalar_t res;      /* result of negotiation */
}

/* presentation context item element */

struct t_ap_syn_off {
    t_scalar_t size;   /* length of syntax object identifier contents */
    t_scalar_t offset; /* offset of object identifier for the syntax  */
}

/* values for res of a presentation context item */

#define T_PCL_ACCEPT              0x0000 /* pres. context accepted */
#define T_PCL_USER_REJ            0x0100 /* pres. context rejected */
                                         /* by peer application    */
#define T_PCL_PREJ_RSN_NSPEC      0x0200 /* prov. reject:          */
                                         /* no reason specified    */
#define T_PCL_PREJ_A_SYTX_NSUP    0x0201 /* prov. reject: abstract */
                                         /* syntax not supported   */
#define T_PCL_PREJ_T_SYTX_NSUP    0x0202 /* prov. reject: transfer */
                                         /* syntax not supported   */
```

```
#define T_PCL_PREJ_LMT_DCS_EXCEED 0x0203 /* prov. reject: local    */
                                         /* limit on DCS exceeded  */

/* Reason Codes for Disconnections */

#define T_AC_U_AARE__NONE     0x0001 /*connection rejected by      */
                                     /*peer user: no reason given  */
#define T_AT_C_U_AARE_ACN     0x0002 /*connection rejected:        */
                                     /*application context name    */
                                     /*not supported               */
#define T_AC_U_AARE_APT       0x0003 /*connection rejected:        */
                                     /*AP title not recognised     */
#define T_AC_U_AARE_AEQ       0x0005 /*connection rejected:        */
                                     /*AE qualifier not recognised */
#define T_AC_U_AARE_PEER_AUTH 0x000e /*connection rejected:        */
                                     /*authentication required     */
#define T_AC_P_ABRT_NSPEC     0x0011 /*aborted by peer provider:   */
                                     /*no reason given             */
#define T_AC_P_AARE_VERSION   0x0012 /*connection rejected:        */
                                     /*no common version           */
```

*Appendix G*

# *SNA Transport Provider*

## G.1 Introduction

This Appendix includes:

- Protocol-specific information that is relevant for Systems Network Architecture (SNA) transport providers.

  It assumes native SNA users, that is, those prepared to use SNA addresses and other SNA transport characteristics (for example, mode name for specifying quality of service).

- Information on the mapping of XTI functions to Full Duplex (FDX) LU 6.2.

  Systems that do not support LU 6.2 full duplex can simulate them using twin-opposed half-duplex conversations. Protocols for doing so will be published separately.

  The half-duplex verbs have been published for several years. The full duplex verbs are documented in the CPI-C Reference, Version 2.1[19].

This Appendix also defines data structures and constants required for NetBIOS transport providers which are exposed through **<xti_sna.h>** header file.

**Note:** Applications written to compilation environments earlier than those required by this issue of the specification (see Section 1.3 on page 3) and defining _XOPEN_SOURCE to be less than 500, may have these data structures and constants exposed through the inclusion of **<xti.h>**.

19. CPI-C Reference, Version 2.1; publicly available from IBM branch offices under the IBM reference SC26-4399.

## G.2 SNA Transport Protocol Information

This section describes the protocol-specific information that is relevant for Systems Network Architecture (SNA) transport providers.

### G.2.1 General

1. Protocol address

   For information about SNA addresses, see Section G.2.2 on page 355.

2. Connection establishment

   Native SNA has no confirmed allocation protocol for full duplex conversations. When a conversation is allocated, the connection message is buffered and sent with the first data that is sent on the conversation. When the *t_connect*( ) or *t_rcvconnect*( ) function completes, connectivity has been established to the partner node, but not to the partner program. Since notification that the partner is not available may occur later, the disconnection reasons returned on *t_rcvdis*( ) include [T_SNA_CONNECTION_SETUP_FAILURE], indicating that the connection establishment never completed successfully.

   An SNA program that needs to know that the partner is up and running before it proceeds sending data must have its own user-level protocol to determine if this is so.

3. Parallel connections

   LU 6.2 allows multiple, simultaneous connections between the same pair of addresses. The number of connections possible between two systems depends on limits defined by system administrators.

4. Sending data of zero octets is supported.

5. Expedited data

   In connection-oriented mode, expedited data transfer can be negotiated by the two transport providers during connection establishment. Expedited data transfer is supported if both transport providers support it. However negotiation between transport users is not supported. Therefore the expedited option is read-only.

6. *t_close*( )

   The semantics of *t_close*( ) on an SNA Transport Provider is simplex orderly, that is, the send pipe of the XTI application issuing the *t_close*( ) is closed, but the receive pipe remains open. Any data sent prior to the *t_close*( ) will be delivered to the partner.

7. SNA buffers data from multiple *t_snd*( ) functions until the SNA send buffer is full, allowing multiple records to be sent in one transmission. However, users sometimes have reasons for ensuring that a record is sent immediately. By setting the T_PUSH flag on the *t_snd*( ) function, the transport user causes data to be transferred without waiting for the buffer to be filled.

   In order to take advantage of the performance improvement that SNA buffering offers, the XTI user must set the T_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.

8. Programs migrated to SNA from other transport providers may want every *t_snd*( ) to cause a message to be sent immediately in order to match behaviour on the original provider. The default of this option is T_YES; thus the default is that a *t_snd*( ) will always be sent out immediately.

### G.2.2 SNA Addresses

In an SNA environment, the protocol address always includes a network-ID-qualified logical unit (LU) name. This is the address of the node where the program resides.

For the *t_connect*( ) and *t_sndudata*( ) functions, the address also contains a transaction program name (TPN), identifying the program addressed in the partner node. A file descriptor used to accept incoming connection requests should have a complete SNA name, including TPN, bound to it with *t_bind*( ).

A file descriptor used for outgoing connection requests may optionally have only a network-id-qualified LU name bound to it.

Since the *t_listen*( ) returns only the LU name part of the address, this address is not adequate for opening up a connection back to the source. The transport user must know the TPN of its partner by some mechanism other than XTI services.

However, *t_rcvudata*( ) returns the complete address of the partner that can be used to send a datagram back to it.

An SNA address has the following structure. When the TPN is not included, the TPN length (sna_tpn_length) is set to zero, and the string that follows is null.

```
/* The definitions for maximum LU name and netid lengths have specific  */
/* values because these maxima are a fixed SNA characteristic,          */
/* not an implementation option.  Maximum TP length is a implementation */
/* option, although the maximum maximum is 64.                          */

#define T_SNA_MAX_NETID_LEN   8
#define T_SNA_MAX_LU_LEN      8
#define T_SNA_MAX_TPN_LEN

struct sna_addr{
    u_char  sna_netid    (T_SNA_MAX_NETID_LEN),
    u_char  sna_lu       (T_SNA_MAX_LU_LEN),
    u_short sna_tpn_len,    /* less than or equal to T_SNA_MAX_TPN_LEN */
    u_char  sna_tpn      (sna_tpn_len)
   }
```

**Notes**:

1. network-identifier (sna_netid): The address can contain either an SNA network identifier or the defined value, SYS_NET, which indicates that the predefined network identifier associated with the local system should be used.
2. IBM Corporation provides a registration facility for SNA network identifiers to guarantee global uniqueness. (See IBM document G325-6025-0, SNA Network Registry).
3. LU name (sna_lu): The address can contain either a specific LU name or the defined value, SYS_LU, which indicates that the system default LU name is to be used.
4. LU name and network identifier fields are fixed length. For values shorter than 8 characters, they are blank filled to the right.
5. Transaction program name (sna_tpn): This field can take one of three values:
   — **Null value**: No transaction program name is to be associated with the file descriptor.

This is adequate for file descriptors used for outgoing connection requests.

If no transaction program is associated with a file descriptor when a *t_listen*( ), *t_rcvudata*( ), or *t_sndudata*( ) is issued, the function will return a TPROTO error.

— **Specified value**: A value that will be known by a partner program; for example, a well-known transaction program name used by a server.

— **Defined value, DYNAMIC_TPN**: An indication that the system should generate a TP name for the file descriptor.

6. The values SYS_NET, SYS_LU and DYNAMIC_TPN may not be used as real values of the sna_netid, sna_lu or sna_tpn fields, respectively.

### G.2.3 Options

Options are formatted according to the structure **t_opthdr** as described in Chapter 13. A transport provider compliant to this specification supports none, all, or any subset of the options defined in Section G.2.3.1.

#### *G.2.3.1 Connection-Mode Service Options*

The protocol level of all subsequent options is SNA.

All options have end-to-end significance. Some may be negotiated in the XTI states T_IDLE and T_INCON, and all are read-only in all other states except T_UNINIT.

### Options for Service Quality and Expedited Data

Table G-1 shows the SNA options that affect the quality of a connection and the transport service level provided.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_SNA_MODE | `char` | T_SNA_BATCH<br>T_SNA_BATCHSC<br>T_SNA_INTER<br>T_SNA_INTERSC<br>T_SNA_DEFAULT<br>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.<br>If not specified, the default mode is T_SNA_DEFAULT.<br>The default mode characteristics may vary from system to system. |
| T_SNA_ALWAYS_PUSH | `t_uscalar_t` | T_YES / T_NO | If T_YES, every *t_snd*( ) operation will cause the message to be sent immediately.<br>If T_NO, the data from a *t_snd*( ) operation may be buffered and sent later. The transport user can set the T_PUSH flag on a *t_snd*( ) function call to cause the data to be sent immediately.<br>Default value is T_NO.<br>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 T_SNA_ALWAYS_PUSH to T_YES may affect its performance. |

**Table G-1** SNA Options

### G.2.4 Functions

*t_accept*( ) Since user data is not exchanged during connection establishment, the parameter *call→udata.len* must be 0.

*t_bind*( ) The *addr* field of the *t_bind* structure represents the local network-id-qualified LU name of the local logical unit and the transaction program name of the program issuing the *t_bind*( ) function.

If the endpoint was bound in the passive mode (that is, qlen > 0) and the requested address has a null transaction program subfield, the function completes with the T_BADADDR error.

*t_connect*( ) The *sndcall→addr* specifies the network-ID-qualified LU name and transaction program name of the remote connection partner.

An SNA transport provider allows more than one connection between the same address pair.

Since user data cannot be exchanged during the connection establishment phase, *sndcall→udata.len* must be set to 0. On return, *rcvcall→udata.maxlen* should be set to 0.

*t_getinfo*( ) In all states except T_DATAXFER, the function *t_getinfo*( ) returns in the parameter *info* the same information that was returned by *t_open*( ). In T_DATAXFER state, however, the information returned may differ from that returned by *t_open*( ), depending on whether the remote transport provider supports expedited data transfer. The fields of *info* are set as defined in the table below.

| Parameters | Before Call | After Call |
|---|---|---|
| fd | x | / |
| info→addr | / | 82 |
| info→options | / | x [1] |
| info→tsdu | / | T_INFINITE (–1) |
| info→etsdu | / | T_INVALID (–2) / 86 [2] |
| info→connect | / | T_INVALID (–2) |
| info→discon | / | T_INVALID (–2) |
| info→servtype | / | T_COTS_ORD |
| info→flags | / | T_SNDZERO |

**Table G-2** Fields for *info* Parameter

**Notes:**

1. x means an integral number greater than zero.
2. Depending on the negotiation of expedited data transfer.

*t_getprotaddr*( ) The *boundaddr* value includes the transaction program name of the local program.

The *peeraddr* value (if any) includes only the network-ID-qualified LU name of the partner.

*t_listen*( ) The *call→addr* structure contains the network-ID-qualified LU name of the remote partner.

*t_open*( ) The default characteristics returned by *t_open*( ) are shown in the table below.

| Parameters | Before Call | After Call |
|---|---|---|
| name | x | / |
| oflag | x | / |
| info→addr | / | 82 |
| info→options | / | x [1] |
| info→tsdu | / | T_INFINITE (–1) |
| info→etsdu | / | T_INVALID (–2) / 86 [2] |
| info→connect | / | T_INVALID (–2) |
| info→discon | / | T_INVALID (–2) |
| info→servtype | / | T_COTS_ORD |
| info→flags | / | T_SNDZERO |

**Table G-3** Default Characteristics returned by *t_open*( )

**Notes:**

1. x means an integral number greater than 0.
2. Expedited data may or may not be supported by the local transport provider.

*t_rcv*( ) 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.

*t_rcvconnect*( ) Since no user data can be returned on *t_rcvconnect*( ), the *call→udata.len* should be set to 0 before the function is invoked.

*t_rcvdis*( ) Since user data is not sent during disconnection, the value *discon→udata.len* should be set to 0 before t_rcvdis() is called.

The following disconnection reason codes are valid for any implementation of an SNA transport provider under XTI:

```
#define T_SNA_CONNECTION_SETUP_FAILURE.
#define T_SNA_USER_DISCONNECT
#define T_SNA_SYSTEM_DISCONNECT
#define T_SNA_TIMEOUT
#define T_SNA_CONNECTION_OUTAGE
```

These definitions are exposed by the inclusion of **<xti_sna.h>**.

*t_snd*( ) Unless the T_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.

In order to take advantage of the performance improvement that SNA buffering offers, the XTI user must set the T_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.

*t_snddis*( ) Since no user data is sent during a disconnection operation, *call→udata.len* should be set to 0 before the call to *t_snddis*( ).

*t_sndudata*( )

The *unitdata→addr* field contains the full SNA address, including network-id-qualified LU name and transaction program identifier, of the remote partner.

If address associated with the file descriptor has a null transaction program name subfield, the function completes with the TPROTO error.

The *unitdata→opt* structure may contain an SNA mode governing the transmission of the data. For example, the program may confine data transmission to secure lines by selecting the T_SNA_INTERSC or T_SNA_BATCHSC modes.

## G.3 Mapping XTI to SNA Transport Provider

This section presents the mapping of XTI functions to Full Duplex (FDX) LU 6.2.

First, several flow diagrams are given to illustrate the function of the XTI Mapper. Following this, mapping tables are given that show the FDX LU 6.2 verbs that the XTI Mapper needs to generate for each XTI function. For each XTI function that maps to a FDX verb, an additional table is referenced that gives the mappings of each parameter. Finally, a table shows mapping of LU 6.2 FDX return codes to XTI events.

The use of FDX LU 6.2 verbs in this section is for illustrative purposes only and is analogous to OSI's use of service primitives, that is, as a way to explain the semantics provided by the protocol. The FDX LU6.2 verbs are used only to help in understanding the SNA protocol, and are not a required part of an implementation.

### G.3.1 General Guidelines

General guidelines for mapping XTI to an SNA transport provider are listed below:

- In the following flow diagrams, notice that the XTI mapper always has a RECEIVE_AND_WAIT posted. This is done so that when data comes into the SNA transport provider, the XTI mapper is able to set an event indicator. Then, when a T_LOOK is issued, the XTI application can be informed that there is data to be received.
- The XTI mapper keeps a table that maps an XTI fd to a RESOURCE(variable) on the FDX verbs.
- In this section, we assume the XTI mapper will be using the FDX LU 6.2 basic conversation verb interface. The following table Table G-4 gives an explanation for each FDX verb that is used in these mappings.

**Table G-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.<br>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. |

### G.3.2 Flows Illustrating Full Duplex Mapping

The following diagrams show mappings from the XTI function calls for active connection establishment to SNA verb sequences. The first Figure G-1 is used for blocking XTI calls; the second Figure G-2 is used for non-blocking calls.

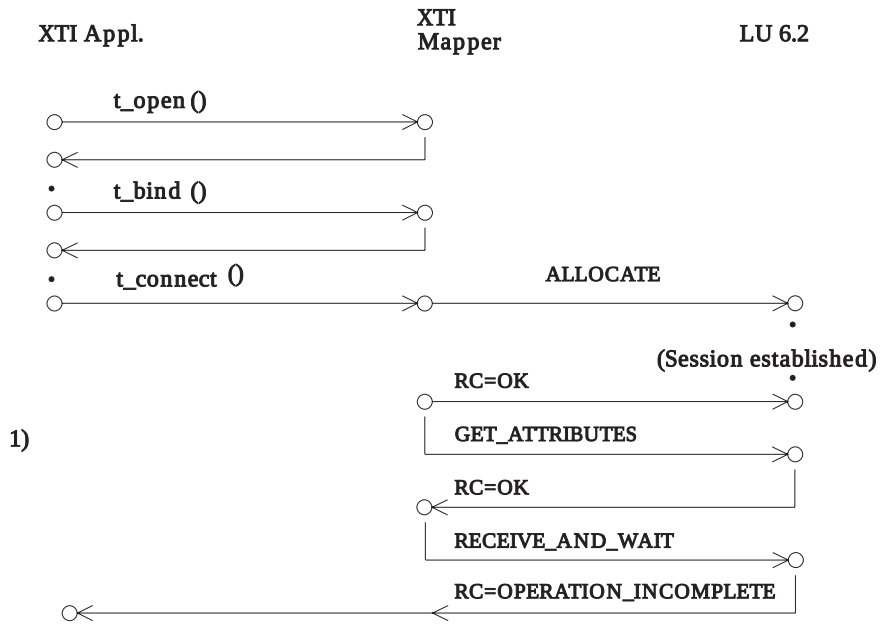


**Figure G-1** Active Connection Establishment, Blocking Version (1 of 2)

#### Annotations

1. GET_ATTRIBUTES is issued after the session is established and before the return for the *t_connect*. This is only done if the mode name, or partner LU name are required on the return to *t_connect*. This would be indicated by a non-zero value in either the *rcvcall→addr.buf* or *eercvcall→opt.buf* fields on *t_connect*.

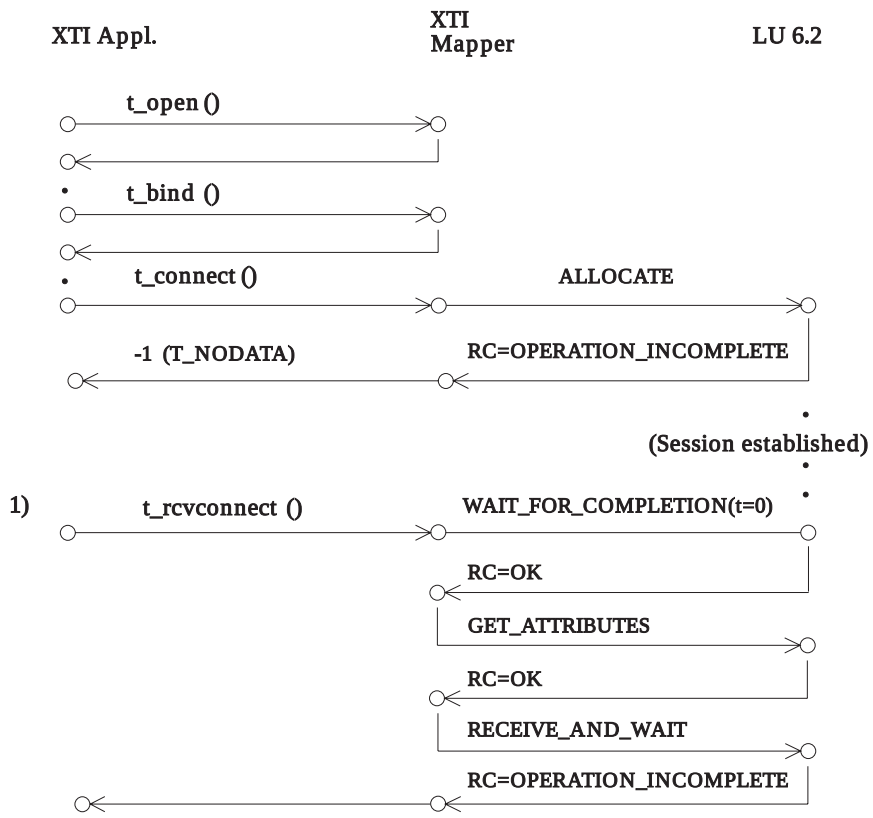


**Figure G-2** Active Connection Establishment, Non-blocking Version (2 of 2)

**Annotations**

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.

   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*.

   After the GET_ATTRIBUTES, a non-blocking RECEIVE_AND_WAIT is issued to post a receive for any incoming data.

The next three diagrams show possible mappings of SNA Attach processing for an incoming connection to the series of XTI calls on the passive side of a connection.

The first Figure G-3 uses the native SNA instantiation mechanism; that is, programs are instantiated when the connection request arrives. This requires that the TP name (that is, the XTI application name) is known as part of the LU definition. This is **before** the *t_bind* is issued.

The second Figure G-4 is a blocking use of the interface, where the SNA transport provider allows a connection request to be received by an existing program. This model, although not described in the architecture, is supported by many SNA products.

The third Figure G-5 is a non-blocking use of the interface, where the SNA transport provider allows a connection request to be received by an existing program. This model is described as part of the FDX architecture.

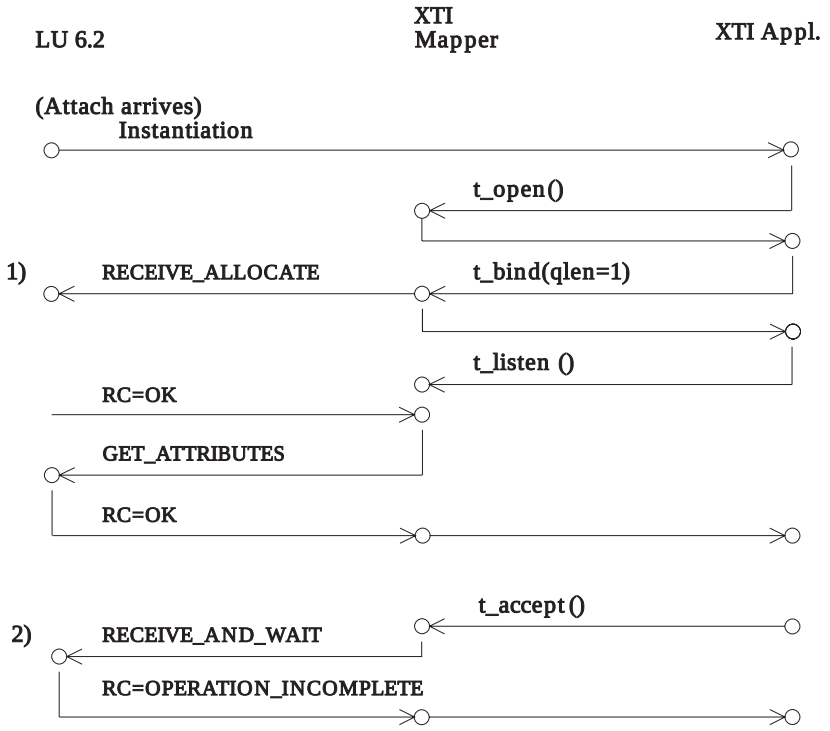


**Figure G-3** Passive Connection Establishment, Instantiation Version (1 of 3)

**Annotations**

1. If qlen in *t_bind* is > 0, a RECEIVE_ALLOCATE will be issued for each connection request that can be queued. 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*.
2. 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.

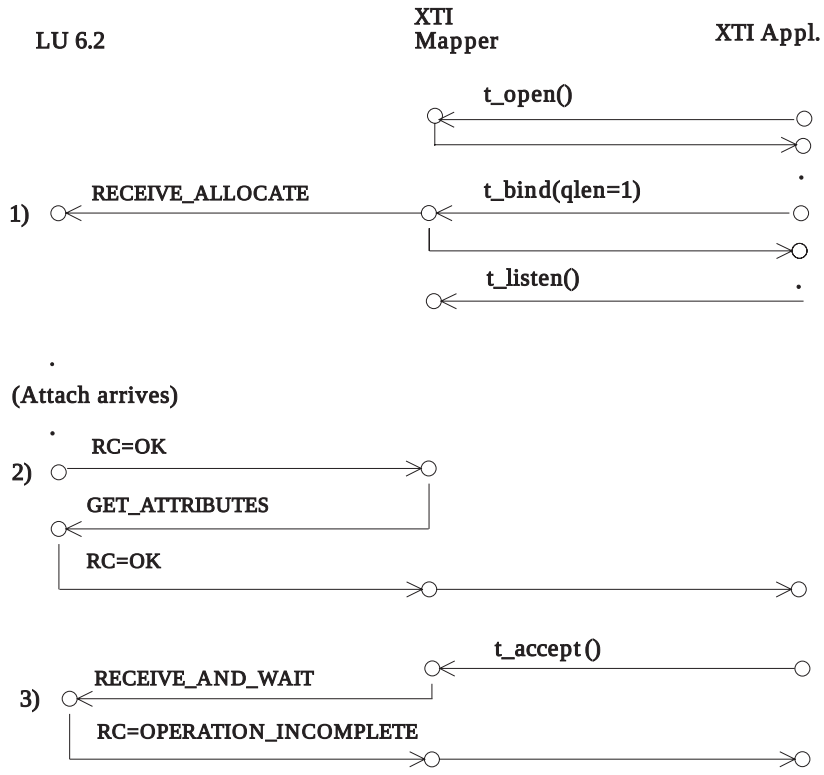


**Figure G-4** Passive Connection Establishment, Blocking Version (2 of 3)

**Annotations**

1. The *t_bind* will cause a blocking RECEIVE_ALLOCATE to be issued for each connection request that can be queued.
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*.
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.

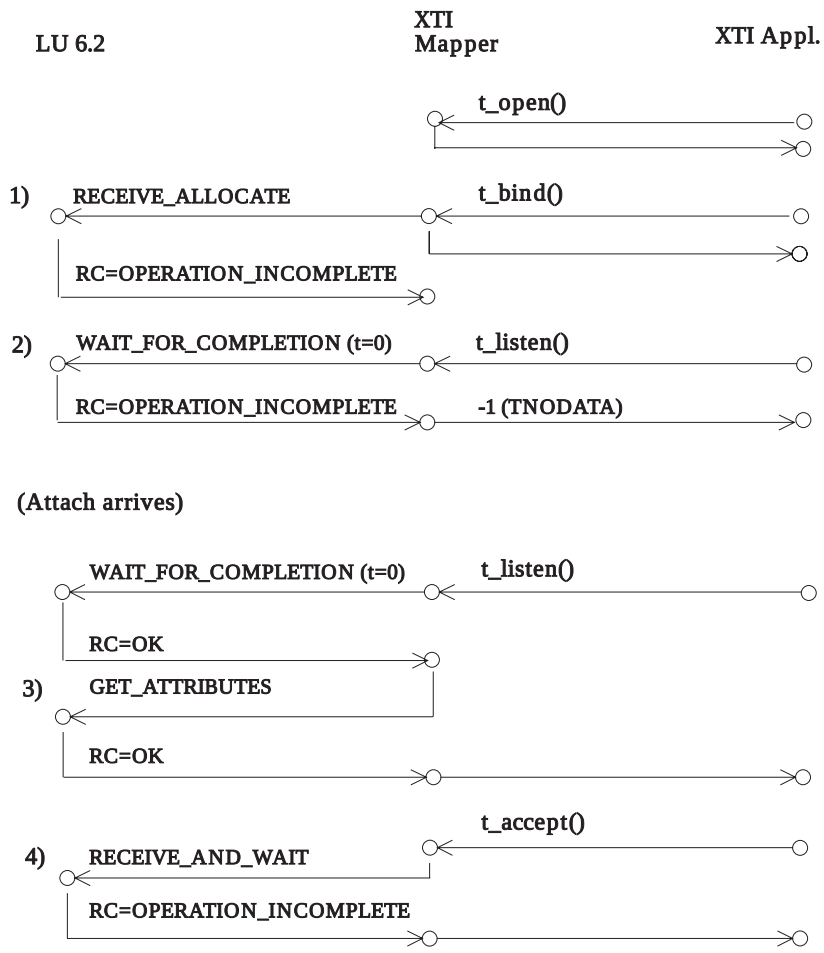


**Figure G-5** Passive Connection Establishment, Non-blocking Version (3 of 3)

**Annotations**

1. The *t_bind* will cause a non-blocking RECEIVE_ALLOCATE to be issued for each connection request that can be queued.
2. A *t_listen* is used as a poll to see if a connection 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.
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*.
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.

The next diagram, Figure G-6, shows the mapping for the blocking XTI *t_snd*( ) call. The diagram after this, Figure G-7, shows the non-blocking mapping of the XTI *t_snd*( ) call.

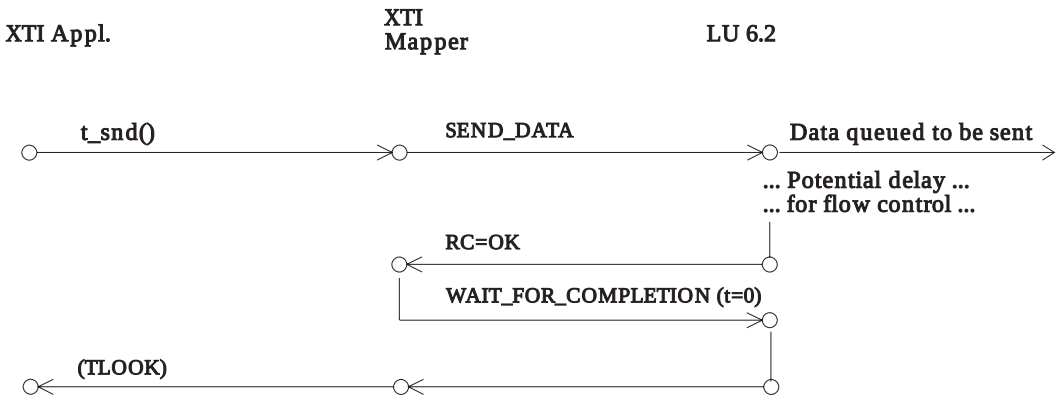


**Figure G-6** XTI Function to LU 6.2 Verb Mapping: Blocking t_snd

1. The blocking *t_snd* will cause a blocking SEND_DATA to be issued. This will block until the LU accepts and queues all the data being sent.

   If EXPEDITED=YES, the mapper will issue a SEND_EXPEDITED_DATA verb rather than the SEND_DATA.

2. When the SEND_DATA returns, 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. At a minimum, there will be an outstanding RECEIVE_AND_WAIT, waiting for any incoming data, that needs to be checked. 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.

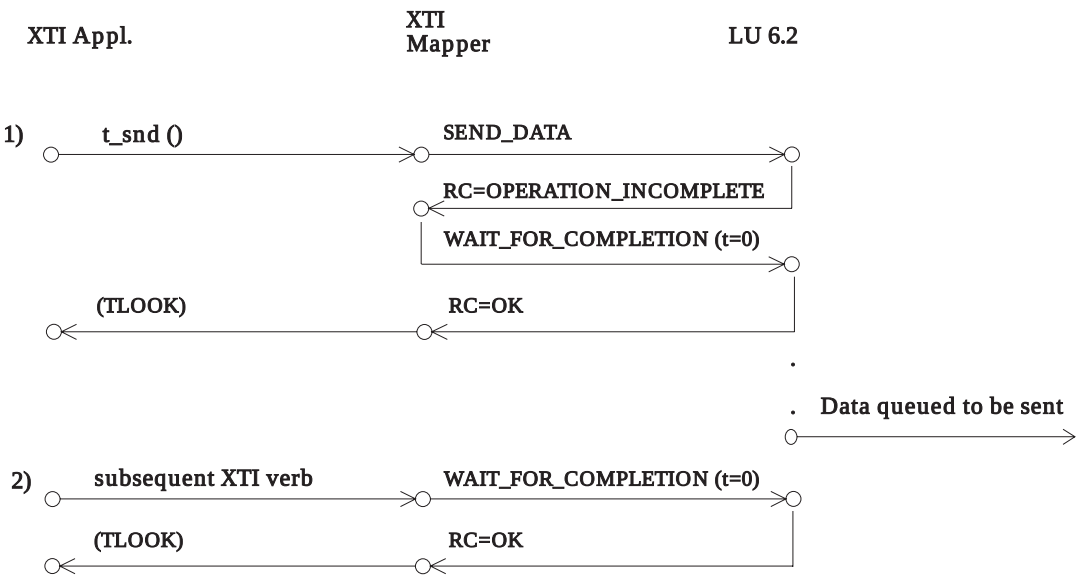


**Figure G-7** XTI Function to LU 6.2 Verb Mapping: Non-blocking t_snd

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.

   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.

   If EXPEDITED=YES, the mapper will issue a SEND_EXPEDITED_DATA verb rather than the SEND_DATA.

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.

   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.

The next diagram, Figure G-8, shows the mapping for blocking XTI receive call, and the diagram after this, Figure G-9, shows the mapping for non-blocking XTI receive call.

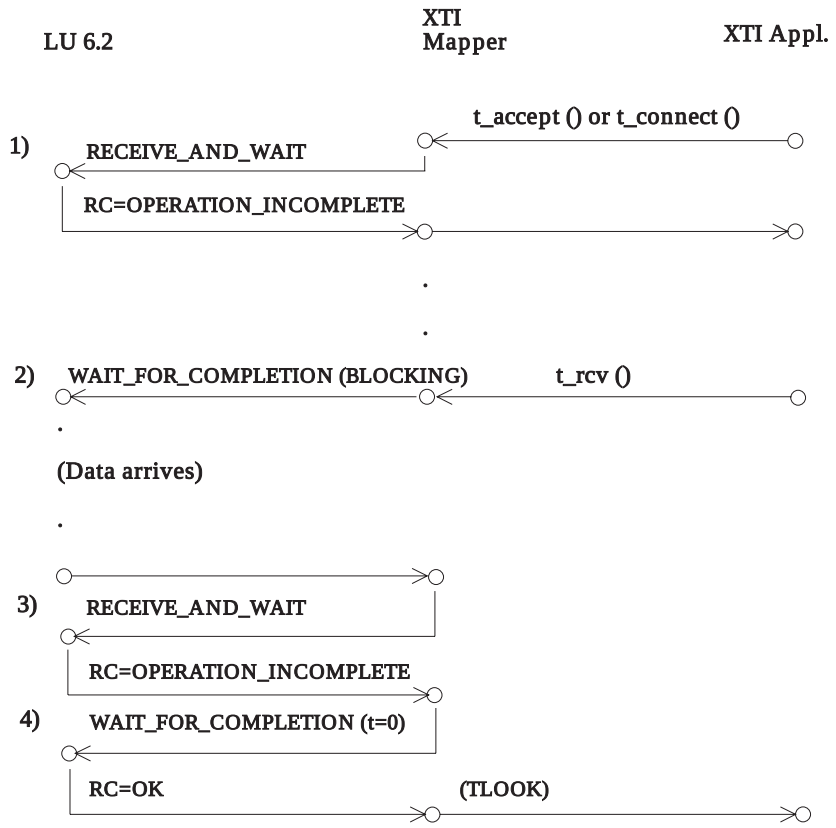


**Figure G-8** XTI Function to LU 6.2 Verb Mapping: Blocking t_rcv

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.

   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.

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.

3. When data is received, the mapper needs to issue a non-blocking RECEIVE_AND_WAIT to replace the one that just completed.

4. Issue a WAIT_FOR_COMPLETION, with time=0, to see if any other outstanding wait objects have been posted.

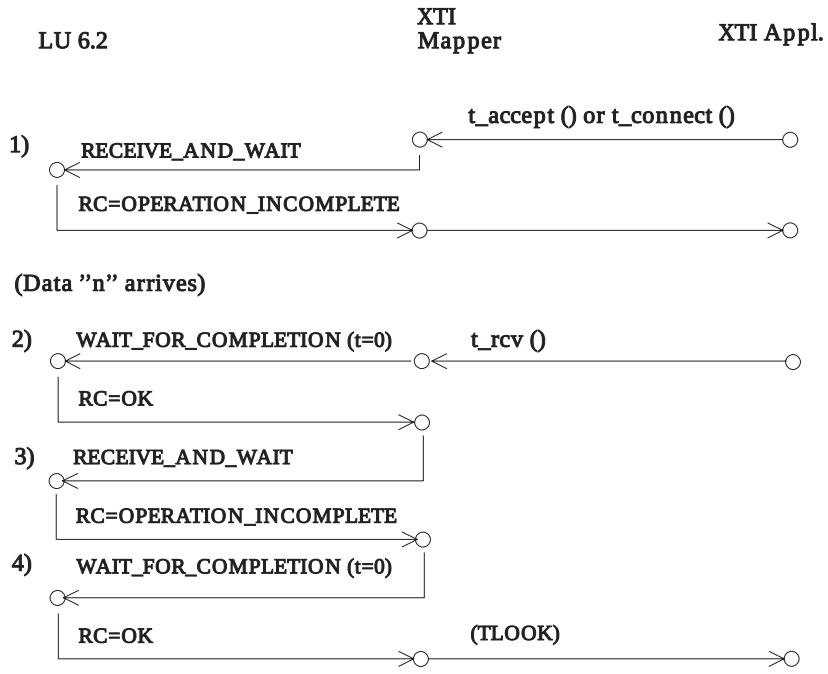


**Figure G-9** Mapping from XTI Calls to LU 6.2 Verbs (Passive side)

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.

   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*.

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.

   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.

3. When data is received, the mapper needs to issue a non-blocking RECEIVE_AND_WAIT to replace the one that just completed.

4. Issue a WAIT_FOR_COMPLETION, with time=0, to see if any other outstanding wait objects have been posted.

### G.3.3 Full Duplex Mapping

The following table shows the mapping from XTI function calls to full duplex LU 6.2 verbs.

**Table G-5** XTI Mapping to LU 6.2 Full Duplex Verbs

| XTI Function | SNA FDX LU6.2 verb | Comments |
|---|---|---|
| *t_accept*( ) | RECEIVE_AND_WAIT | User data is not exchanged during connection establishment.<br>Refer to Table G-7 on page 374. |
| *t_alloc*( ) | Local | |
| *t_bind*( ) | If qlen>0: RECEIVE_ALLOCATE for each connection request that can be queued.<br>Optionally: DEFINE_TP<br>With the instantiation model, the TP name (that is, XTI application name) must be known by the LU before the TP can be instantiated. This is prior to the t_bind being issued. (Refer to Figure G-3 on page 365.) | Refer to Table G-8 on page 375. |
| *t_close*( ) | If connection still up issue DEALLOCATE TYPE(FLUSH) | Refer to Table G-9 on page 375. |
| *t_connect*( ) | ALLOCATE RETURN_CONTROL<br>GET_ATTRIBUTES<br>RECEIVE_AND_WAIT | Refer to Table G-10 on page 376. |
| *t_error*( ) | Local | |
| *t_free*( ) | Local | |
| *t_getinfo*( ) | Local | |
| *t_getprotaddr*( ) | GET_TP_PROPERTIES to get OWN_FULLY_ QUALIFIED_LU_ NAME and OWN_TP_NAME<br>GET_ATTRIBUTES to get PARTNER_FULLY_ QUALIFIED_LU_ NAME | The partner's TP name must be learned by some mechanism other than XTI services. In connectionless mode, there is no partner name.<br>Refer to Table G-11 on page 378. |
| *t_getstate*( ) | Local | |
| *t_listen*( ) | WAIT_FOR_COMPLETION | Refer to Table G-12 on page 379. |
| *t_look*( ) | WAIT_FOR_COMPLETION | |
| *t_open*( ) | Local | *t_open* sets blocking mode (that is, blocking or non-blocking) |
| *t_optmgmt*( ) | GET_ATTRIBUTES | To get Mode name<br>Refer to Table G-13 on page 379. |
| *t_rcv*( ) | WAIT_FOR_COMPLETION<br>RECEIVE_AND_WAIT<br>[RECEIVE_EXPEDITED_DATA]<br>WAIT_FOR_COMPLETION | Refer to Table G-14 on page 380. |

| XTI Function | SNA FDX LU6.2 verb | Comments |
|---|---|---|
| *t_rcvconnect*( ) | WAIT_FOR_COMPLETION<br>GET_ATTRIBUTES<br>RECEIVE_AND_WAIT | Refer to Figure G-2 on page 364. |
| *t_rcvdis*( ) | Local | Event caused by DEALLOCATE_ABEND_* or RESOURCE_FAILURE_* return code on any verb |
| *t_rcvrel*( ) | Local | Event caused by DEALLOCATE_NORMAL return code on RECEIVE_* verb |
| *t_snd*( ) | SEND_DATA (expedited data)<br>[FLUSH]<br>[SEND_EXPEDITED_DATA] | Every *t_snd* causes a SEND_DATA to be issued - even if T_MORE set. If T_MORE is set, the LL continuation bit is set.<br>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.<br>Refer to Table G-16 on page 382. |
| *t_snddis*( ) | DEALLOCATE TYPE(ABEND_PROG) | Takes down both directions of the connection<br>Refer to Table G-17 on page 383. |
| *t_sndrel*( ) | DEALLOCATE TYPE(FLUSH) | Takes down send direction of conversation only.<br>Refer to Table G-19 on page 383. |
| *t_sndudata*( ) | SEND_DATA on datagram server conversation | Refer to Table G-19 on page 383. |
| *t_strerror*( ) | local | |
| *t_sync*( ) | local | |
| *t_unbind*( ) | local | |

*G.3.3.1 Parameter Mappings*

**Table G-6** Relation Symbol Description

| **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. |

**Table G-7** *t_accept* ↔ FDX Verbs and Parameters

| **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) |

**Table G-8** *t_bind* ↔ FDX Verbs and Parameters

| **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)<br>TP_NAME(variable) |
| req→qlen | Used Locally | >0, RECEIVE_ALLOCATE issued for each connection 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)<br>TP_AL_LU_NAME(variable) |
| ret→addr.qlen | Created Locally | |
| | Used Locally | RESOURCE(variable) |
| t_errno | → | RETURN_CODE(variable) |

**Table G-9** *t_close* ↔ FDX Verbs and Parameters

| **XTI Function and Parameters** | ←*Relation*→ | **FDX Verb & Parameter** |
|---|---|---|
| **t_close**<br>**If connection is still in T_DATAXFER state** | | **DEALLOCATE** |
| **Input** | | |
| fd | → | RESOURCE(variable) |
| | Constant | TYPE(FLUSH) |
| **Output** | | |
| t_errno | ← | RETURN_CODE(variable) |

**Table G-10** *t_connect* ↔ FDX Verbs and Parameters

| **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)<br>If platform does not support this tower (Tower 205), use (WHEN_SESSION_ALLOCATED). |
| | Created Locally | WAIT_OBJECT(BLOCKING) if blocking<br>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** | | |
| fd | Used Locally | |

| **XTI Function and Parameters** | ←*Relation*→ | **FDX Verb & Parameter** |
|---|---|---|
| | Created Locally | RESOURCE(variable) |
| **Output** | | |
| | Created Locally | LENGTH(variable) |
| | Created Locally | FILL(addr of local bufr) |
| | Constant | WAIT_OBJECT(BLOCKING) |

**Table G-11** *t_getprocaddr* ↔ FDX Verbs and Parameters

| **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)<br>OWN_TP_NAME(variable) |
| t_errno | ← | RETURN_CODE(variable) |

**Table G-12** *t_listen* ↔ FDX Verbs and Parameters

| **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.buf) | ← | PARTNER_FULLY_QUALIFIED _LU_NAME(variable) |
| call→opt.len | ← | MODE_NAME(variable) |
| bufr&larrow.(call→opt.buf) | ← | MODE_NAME(variable) |

**Table G-13** *t_optmgmt* ↔ FDX Verbs and Parameters

| **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.buf | Used Locally | |
| req→opt.flags | Used Locally | |
| ret→opt.maxlen | Used Locally | |
| ret→opt.buf | Used Locally | |
| **Output** | | |
| ret→opt.len | ← | MODE_NAME(variable) |
| ret→opt.buf | ← | MODE_NAME(variable) |
| ret→flags | Created Locally | |
| t_errno | ← | RETURN_CODE(variable) |

**Table G-14** *t_rcv* ↔ FDX Verbs and Parameters

| **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.<br>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<br>• T_MORE<br>• T_EXPEDITED=NO/YES | Created Locally | WHAT_RECEIVED(variable)<br>If there is expedited data to be received, a RECEIVE_EXPEDITED_DATA verb will be issued to receive it. |
| 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) |

**Table G-15** *t_rcvconnect* ↔ FDX Verbs and Parameters

| **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) |

**Table G-16** *t_snd* ↔ FDX Verbs and Parameters

| **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<br>• T_EXPEDITED=NO<br>• T_MORE<br>• T_FLUSH | → | LL continuation bit |
| | Created Locally | WAIT_OBJECT(BLOCKING) if blocking<br>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<br>• T_EXPEDITED=YES<br>• T_MORE<br>• T_FLUSH | → | LL continuation bit<br>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 |

**Table G-17** *t_snddis* (Existing Connection) ↔ FDX Verbs and Parameters

| XTI Function and Parameters | ←*Relation*→ | FDX Verb & Parameter |
|---|---|---|
| **t_snddis()**<br>**For existing connection** | | **DEALLOCATE** |
| **Input** | | |
| fd | Used Locally | |
| | Created Locally | RESOURCE(variable) |
| call | Constant | =nullptr |
| | Constant | TYPE(ABEND_PROG) |
| **Output** | | |
| t_errno | ← | RETURN_CODE(variable) |

**Table G-18** *t_snddis* (Incoming Connect Req.) ↔ FDX Verbs and Parameters

| XTI Function and Parameters | ←*Relation*→ | FDX Verb & Parameter |
|---|---|---|
| **t_snddis()**<br>**To reject incoming connection request** | | DEALLOCATE |
| **Input** | | |
| call→sequence | Used Locally | |
| **Output** | | |
| t_errno | ← | RETURN_CODE(variable) |

**Table G-19** *t_sndrel* ↔ FDX Verbs and Parameters

| XTI Function and Parameters | ←*Relation*→ | FDX Verb & Parameter |
|---|---|---|
| **t_sndrel()** | | DEALLOCATE |
| **Input** | | |
| fd | Used Locally | |
| | Created Locally | RESOURCE(variable) |
| | Created Locally | TYPE(FLUSH) |
| **Output** | | |
| t_errno | ← | RETURN_CODE(variable) |

### G.3.4 Half Duplex Mapping

The interface to the SNA transport provider is the FDX LU 6.2 interface. If the SNA transport provider does not support FDX LU 6.2, the FDX verbs can be mapped to two half-duplex LU 6.2 connections, one used to send in each direction. This gives the appearance of a full duplex connection without requiring any conversation direction turn-arounds on the half-duplex conversations.

The mapping of FDX LU 6.2 verbs to two half-duplex connections is described in FDX Kit.

### G.3.5 Return Code to Event Mapping

The following table Table G-20 shows how the return codes on LU 6.2 verbs are mapped to XTI events.

Any return code for which there no mapping given in this table will create a disconnect.

**Table G-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 =<br>• DATA_COMPLETE<br>• DATA_INCOMPLETE |
| T_DISCONNECT | One of the following has occurred:<br>SEND_DATA issued with RETURN_CODE = ERROR_INDICATION with subcode from list below:<br>• ALLOCATION_ERROR<br>• DEALLOCATE_ABEND_PROG<br>• DEALLOCATE_ABEND_SVC<br>• DEALLOCATE_ABEND_TIMER<br>• RESOURCE_FAILURE_NO_RETRY<br>• RESOURCE_FAILURE_RETRY<br>Any other verb issued with RETURN_CODE of<br>• ALLOCATION_ERROR<br>• DEALLOCATE_ABEND_PROG<br>• DEALLOCATE_ABEND_SVC<br>• DEALLOCATE_ABEND_TIMER<br>• RESOURCE_FAILURE_NO_RETRY<br>• 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 *t_listen*( ) function call.<br>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. |

## G.4 Compatibility.

Certain symbols may be exposed to applications including **<xti_sna.h>** for compatibility with applications transitioning from older issues of this specification where their semantics are specified. Exposing these symbols is allowed but not required. Symbols that may be exposed in this implementation-dependent manner are:

SNA_MAX_NETID_LEN, SNA_MAX_LU_LEN, SNA_MAX_TPN_LEN SNA_CONNECTION_SETUP_FAILURE, SNA_USER_DISCONNECT, SNA_SYSTEM_DISCONNECT, SNA_TIMEOUT, SNA_CONNECTION_OUTAGE

*Appendix H*

# *IPX/SPX Transport Provider*

## H.1 General

This appendix specifies protocol-specific information that is relevant for mapping XTI functions to SPX and IPX transport providers.

The description given here is limited to the IPX protocol and the enhanced SPX (or SPXII) protocol. All references to the SPX protocol refer to this version unless specifically noted. In compliance with the X/Open Interface Adoption Criteria, this protocol is obtainable from multiple sources.

**Notes:**

1. Neither IPX nor SPX supports expedited data. All data is handled on a first-come, first-served basis.
2. The protocol-specific data structures used by IPX and SPX (most notably, the **T_SPX2_OPTIONS** structure) are likely to grow in the future.

## H.2 Namespace

### H.2.1 IPX

If the header **xti_ipx.h** is included, identifiers with the prefixes, suffixes or complete names shown are reserved for any use by the implementation.

| Header | Prefix |
|---|---|
| **<xti_ipx.h>** | t_ipx |

If the header **xti_ipx.h** is included, macros with the following prefixes may be defined. After the last inclusion of **xti_ipx.h** an application may use identifiers with the following prefixes for its own purpose, provided their use is preceded by an **#undef** of the macro.

| Header | Prefix |
|---|---|
| **<xti_ipx.h>** | T_IPX_ |

### H.2.2 SPX

If the header **xti_spx.h** is included identifiers with the prefixes, suffixes or complete names shown are reserved for any use by the implementation.

| Header | Prefix |
|---|---|
| **<xti_spx.h>** | t_spx, T_SPX versionNumber |

If the header **xti_spx.h** is included, macros with the prefixes may be defined. After the last inclusion of **xti_spx.h** an application may use identifiers with the following prefixes for its own purpose, provided there use is preceded by an **#undef** of the macro.

| Header | Prefix |
|---|---|
| **<xti_spx.h>** | T_SPX_ |

## H.3 Options

### H.3.1 IPX-level Options

IPX options are association related. IPX options may be negotiated in all XTI states except T_UNBND and T_UNINIT. The level associated with each option is T_IPX_OPT. The name member should be set to T_IPX_OPTS_V1. IPX options are stored in a structure of type **t_ipxOptions** that follows the XTI **t_opthdr** structure containing at least the following members:

```
typedef struct t_ipxOptions {
    unsigned short    ipx_checksum       /* Checksum        */
    unsigned char     ipx_packet_type    /* IPX Packet type */
    } t_ipxOpts_t;
```

This option may only be manipulated using the *t_sndudata*( ) or *t_rcvudata*( ) functions.

Other options may be defined in the future.

At least the following packet types are known to IPX:

T_IPX_NULL_PACKET_TYPE Used for all packets not classified by any other type

T_IPX_NCP_PACKET_TYPE Used for NCP packets (that is, Netware Control Protocol)

T_IPX_SPX_PACKET_TYPE Sequenced packet protocol used for SPX packets

### H.3.2 SPX-level Options

Some values of the SPX options have end-to-end significance as defined below. Unless otherwise noted, they may be negotiated in all XTI states except T_UNBND and T_UNINIT. The level associated with each option is T_SPX_OPT. The name member is set to T_SPX_OPTS_V1. SPX options are stored following the XTI **t_opthdr** structure in a format defined by the following data structure:

```
typedef struct t_spx2_options {
    unsigned int    versionNumber;
    unsigned int    spxIIOptionNegotiate;
    unsigned int    spxIIRetryCount;
    unsigned int    spxIIMinimumRetryDelay;
    unsigned int    spxIIMaximumRetryDelta;
    unsigned int    spxIIWatchdogTimeout;
    unsigned int    spxIIConnectionTimeout;
    unsigned int    spxIILocalWindowSize;
    unsigned int    spxIIRemoteWindowSize;
    unsigned int    spxIIConnectionID;
    unsigned int    spxIIInboundPacketSize;
    unsigned int    spxIIOutboundPacketSize;
    unsigned int    spxIISessionFlags;
    } T_SPX2_OPTIONS;
```

An application passing a **t_opthdr** structure followed by a **T_SPX2_OPTIONS** structure to a function that can support options information will get information about all legal options on each call.

Each of this structure's members is described below.

versionNumber
: The application must set this member to T_SPX_OPTIONS_VERSION. This member is used to manage forward compatibility as this structure is extended in the future. Access is through the *t_optmgmt*( ) or other functions that use the **t_call** data structure. T_SPX_OPTIONS_VERSION is currently set to 1.

spxIIOptionNegotiate
: This value is association-related (that is, has end-to-end significance). This member can be set to T_SPX_NEGOTIATE_OPTIONS (default) to indicate that an endpoint wishes to exchange option data with a remote endpoint, or to T_SPX_NO_NEGOTIATE_OPTIONS to indicate that it does not wish to do this. This is a negotiable value, and may be manipulated through the *t_optmgmt*( ) function or through functions that use the **t_call** data structure.

spxIIRetryCount
: This value specifies the number of unsuccessful transmission attempts that SPX will make before aborting a connection. Setting this value to 0 causes the default value (10) to be used. This is a negotiable value, and may be manipulated through the *t_optmgmt*( ) function or through functions that use the **t_call** data structure.

spxIIMinimumRetryDelay
: An internal round-trip time algorithm normally calculates the delay before a transmission retry. Setting this member to a non-zero value overrides this algorithm and uses the value as the fixed number of milliseconds before a retransmit. The default value is 300 milliseconds. This is a negotiable value, and may be manipulated through the *t_optmgmt*( ) function or through functions that use the **t_call** data structure.

spxIIMaximumRetryDelta

The value of this member is added to *spxIIMinimumRetryDelay* or to the current round-trip time to determine the maximum retry delay. Setting this value to 0 causes the default delay (5 seconds) to be used. This is a negotiable value, and may be manipulated through the *t_optmgmt*( ) function or through functions that use the **t_call** data structure.

spxIIWatchdogTimeout

This value determines the number of seconds that SPX will wait on an inactive connection before sending a still-alive query to the remote endpoint. This is a negotiable value, and may be manipulated through the *t_optmgmt*( ) function or through functions that use the **t_call** data structure.

spxIIConnectionTimeout

This value is the number of seconds that SPX will wait after a successful connect request before the first session packet must arrive. If a packet does not arrive in this interval, the connection is aborted. This is a negotiable value, and may be manipulated through the *t_optmgmt*( ) function or through functions that use the **t_call** data structure.

spxIILocalWindowSize

This value is association-related (that is, has end-to-end significance). This member sets the size of the local endpoint's receive window in packets. A zero setting requests the driver to negotiate the window size. The local driver default is 8. This is a negotiable value, and may be manipulated through the *t_optmgmt*( ) function or through functions that use the **t_call** data structure.

spxIIRemoteWindowSize

This value is association-related (that is, has end-to-end significance). This member is meaningful only after a connection has been established, that is, in states T_OUTCON, T_INCON, T_DATAXFER. It contains the number of packets in the remote endpoint's receive window. This is a non-negotiable value and is retrieved through functions that use the **t_call** data structure. Attempts to set this value with the T_NEGOTIATE flag set will have no effect.

spxIIConnectionID

This value is association-related (that is, has end-to-end significance). This member is meaningful only after a connection has been established, that is, in states T_OUTCON, T_INCON, T_DATAXFER. It contains the local endpoint connection ID. This a non-negotiable value and is retrieved through functions that use the **t_call** data structure. Attempts to set this value with the T_NEGOTIATE flag set will have no effect.

spxIIInboundPacketSize

This value is association-related (that is, has end-to-end significance). This member is meaningful only after a connection has been established, that is, in states T_OUTCON, T_INCON, T_DATAXFER. It contains the number of bytes in each incoming data packet. This value may change due to a route change in mid-connection. There is no way to notify an application that this has occurred. This is a non-negotiable value and is retrieved through functions that use the **t_call** data structure. Attempts to set this value with the T_NEGOTIATE flag set will have no effect.

spxIIOutboundPacketSize

This value is association-related (that is, has end-to-end significance). This member is meaningful only after a connection has been established, that is, in states T_OUTCON, T_INCON, T_DATAXFER. It contains the number of bytes in each outgoing data packet. This value may change due to a route change in mid-connection. There is no way to notify an application that this has occurred. This is a non-negotiable value and is retrieved through functions that use the **t_call** data structure. Attempts to set this value with the

T_NEGOTIATE flag set will have no effect.

spxIISessionFlags

This value is association-related (that is, has end-to-end significance). This bit member contains flags that control physical layer characteristics of SPX packets. These may include checksums, data encryption, or data signing. The following flags have been defined:

| Flag | Value | Description |
|---|---|---|
| T_SPX_SF_NONE | 0x00 | Options off |
| T_SPX_SF_IPX_CHECKSUM | 0x01 | Packet checksums on |
| T_SPX_SF_SPX2_SESSION | 0x02 | Compatibility flag |

The T_SPX_SF_NONE and T_SPX_SF_IPX_CHECKSUM flags are read/write, and accessible through the *t_optmgmt*( ) function and functions using the **t_call** data structure. The T_SPX_SF_SPX2_SESSION flag is read-only and is accessible through functions that use the **t_call** data structure.

## H.4 Functions

*t_accept*( ) No special considerations.

*t_bind*( ) **For IPX and SPX**:

IPX and SPX support static and dynamic port assignment. If the application does not wish to chose a port, it sets *req* to NULL or *req→addr.len* to 0. In this case, the provider assigns a dynamic port in the range 0x4000 to 0x7fff.

If the application wishes to bind to a specific port (that is, a static port), *req→addr.buf* must point to an addressing structure.

A process without sufficient privilege can only request a port in the range 0x8000 to 0xffff. If the request is for a port outside the static port range, *t_bind*( ) will fail and *t_errno* will be set to TACCESS. A process with appropriate privilege can request any port number.

**For SPX**:

SPX allows only a single transport endpoint to be bound to a port. If a requested port is already bound to another endpoint, *t_bind*( ) will fail and set *t_errno* to [TADDRBUSY].

*t_connect*( ) SPX does not support connect user data.

*t_getinfo*( ) The default characteristics returned by *t_getinfo*( ) in the **info** structure for an IPX endpoint are:

| Parameters | Before call | After call |
|---|---|---|
| *info→addr* | / | x |
| *info→options* | / | x |
| *info→tsdu* | / | x (1) |
| *info→etsdu* | / | T_INVALID (–2) |
| *info→connect* | / | T_INVALID (–2) |
| *info→discon* | / | T_INVALID (–2) |
| *info→servtype* | / | T_CLTS |
| *info→flags* | / | 0 |

(1) ''x'' is the smallest of the maximum size of transport data units supported by connected LANs.

The default characteristics returned by *t_getinfo*( ) in the **info** structure for an SPX endpoint are:

| Parameters | Before call | After call |
|---|---|---|
| *info->addr* | / | x |
| *info->options* | / | x |
| *info->tsdu* | / | T_INFINITE (–1) |
| *info->etsdu* | / | T_INVALID (–2) |
| *info->connect* | / | T_INVALID (–2) |
| *info->discon* | / | T_INVALID (–2) |
| *info->servtype* | / | T_COTS_ORD |
| *info->flags* | / | 0 |

*t_listen*( ) SPX does not support connect user data.

An option buffer, as described in Section K.3.2, ''SPX-level Options'' will be passed to the user (unless *call→opt.maxlen* is zero.

*t_open*( ) The default characteristics returned by *t_open*( ) in the **info** structure for an IPX endpoint are the same as those listed in the table above for *t_getinfo*( ).

Similarly, the default characteristics returned by *t_open*( ) in the **info** structure for an SPX endpoint are the same as those listed in the table above for *t_getinfo*( ).

*t_optmgmt*( ) **For IPX**:

No IPX options are currently available through the *t_optmgmt*( ) function.

**For SPX**:

Options may be set or retrieved using the *t_optmgmt*( ) function only when the endpoint is in the state T_IDLE.

SPX supports the standard options buffer using the **t_opthdr** structure. The **t_opthdr** structure is followed by a **T_SPX2_OPTIONS** structure, as described in Section K.3, Options.

SPX supports options that may be examined and negotiated with the *t_optmgmt*( ) function. The options are listed and described in Section K.3, Options.

*t_rcv*( ) SPX does not support expedited data, so the T_EXPEDITED flag will not be set.

SPX allows logical units of data to be unlimited in length.

If the SPX watchdog (see Section K.3.2, ''SPX-level Options'' determines that the remote transport endoint is no longer participating in the connection, the SPX watchdog generates a disconnect indication which causes *t_rcv*( ) to return with a [T_LOOK] error.

*t_rcvdis*( ) SPX does not support disconnect user data.

On return, the *discon→reason* member is set to one of the following:

T_SPX_CONNECTION_TERMINATED
Indicates that no error occurred and an SPX terminate connection packet was received from the remote endpoint. This reason code indicates success.

T_SPX_CONNECTION_FAILED
Indicates that the remote endpoint failed to acknowledge a transmission.

*t_rcvudata*( ) The *t_rcvudata* function is issued on an IPX endpoint.

On return from the function, *unitdata→addr.buf* points to the address information for the remote endpoint, and *unitdata→opt.buf* points to an options buffer as described in Section K.3.1, ''IPX-level Options''.

If a packet received by a *t_rcvudata*( ) request did not have the checksum calculated and verified, the *t_ipx_checksum* member of the **struct t_ipxOptions**, is set to 0xFFFF. Any other value indicates that the provider has calculated and verified the checksum of the received packet. For all received packets, the *t_ipx_packet_type* member is set to the packet type of the received packet.

*t_snd*( ) SPX does not support expedited data so the T_EXPEDITED flag must not be set.

*t_snddis*( ) SPX does not support the sending of user data or options with a disconnect request.

*t_sndudata*( ) If *unitdata→opt.len* is zero, then a packet type of T_IPX_NULL_PACKET_TYPE is used and no checksum is generated.

To send any other packet type, *unitdata->opt* must reference an options buffer as defined in Section K.3.1. ''IPX-level Options''. The IPX packet type must be defined. A checksum will be generated if *t_ipx_checksum*" is set to T_IPX_CHECKSUM_TRIGGER.

*Appendix I*

# *ATM Transport Protocol Information for XTI*

## I.1 General

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

The following notes apply:

- This version of XTI supports a subset of the functions specified by the ATM Forum as the User-Network Interface (see reference **UNI**), versions 3.0 and 3.1.
- Frequent reference is made to the ATM Forum's ''Native ATM Services: Semantic Description, Version 1.0'', identified in this Appendix as reference **ATMNAS**.
- The ATM transport provider does not support Connectionless Transport Service, so use of the functions *t_rcvudata*( ), *t_rcvuderr*( ) and *t_sndudata*( ) always cause the [TNOTSUPPORT] error to be returned.
- The ATM transport provider supports both reliable and unreliable data transport services. This is selected via parameter *name* of function *t_open*( ). Note that both services are categorized as a T_COTS type transport provider.
- The ATM transport provider does not support an orderly release mechanism, so use of the functions *t_sndrel*( ) and *t_rcvrel*( ) always cause the [TNOTSUPPORT] error to be returned.
- The ATM transport provider does not support expedited data transfer.
- The ATM transport provider does not support sending user data during connection setup or release.
- The ATM transport provider does not support ATM PVCs.
- The ATM transport provider does not support a mechanism to specify the congestion indication bit and the user-user byte in AAL5.
- At the current time, only AAL-5 message mode is supported; each TSDU is carried across the network in the payload field of a single AAL-5 PDU. The ATM transport provider does not support AAL1 and User-defined AAL.
- When a transport user passively waits for incoming connect indications through a transport endpoint, the ATM protocol address bound to the endpoint must conform to the ATM Forum guidelines (referenced document **ATMNAS**) for the specification of a SAP address. The SAP address is a vector that includes fields for the ATM network address (with selector byte), identification of a layer-2 protocol, identification of a layer-3 protocol, and identification of an application
- At the current time, alternate BLLI negotiation is not supported; only a single BLLI information element is used in connection setup.
- A new event, T_LEAFCHANGE, is defined which is used by event management to notify a transport user when the status of a leaf has changed on a point-to-multipoint connection. Specifically, event T_LEAFCHANGE is flagged whenever any one of the following occurs:
  - — an attempt to add a new leaf via *t_addleaf*( ) in asynchronous mode was successful
  - — an attempt to add a new leaf via *t_addleaf*( ) in asynchronous mode was unsuccessful

— either the leaf or the ATM network caused the leaf to be removed from the connection.

## I.2 ATM Addresses

This section describes the two kinds of ATM addresses mentioned in this appendix: network and protocol.

### I.2.1 ATM Network Address

An ATM network address specifies an ATM device that resides on the user side of the User Network Interface (UNI). There are three typical uses of an ATM network address:

- A transport user actively initiating a connection can specify the ATM network address from which the connection is originating. This is helpful when the originating device supports multiple ATM network addresses over the same UNI.
- A transport user passively receiving a connection can inspect the ATM network address from which the connection originated.
- A transport user acting as the root of a point-to-multipoint connection specifies the ATM network address of any leafs to be added to the connection.

The ATM network address is expressed by either the **t_atm_addr** structure or the **t_atm_sap** structure, depending on the context. The **t_atm_addr** structure is used to query and negotiate options with the transport provider, whereas the **t_atm_sap** structure is used for any function that requires an ATM address as a parameter.

The **t_atm_addr** structure is defined in the options section. The **t_atm_sap** structure is defined in Section I.2.3 on page 397. Note that when the **t_atm_sap** structure is used to convey a network address, the following fields of the structure must have a value of TATM_ABSENT:

*t_atm_sap_layer2.SVE_tag*
*t_atm_sap_layer3.SVE_tag*
*t_atm_sap_appl.SVE_tag*

### I.2.2 ATM Protocol Address

An ATM protocol address specifies an ATM device that resides on the user side of the UNI, plus a service access point (SAP) within the said ATM device. There are two typical uses of an ATM protocol address:

- A transport user actively initiating a connection specifies an ATM protocol address as the destination of the connection.
- A transport user passively receiving a connection specifies an ATM protocol address as the address at which the transport user is awaiting the incoming connection.

### I.2.3 t_atm_sap Structure

The ATM address (network or protocol) is expressed via the **t_atm_sap** structure, defined below. Note that an implementation may overlay this structure onto a character array, prepended by other information (for example, address family identifier).

```
struct  t_atm_sap {

    struct t_atm_sap_addr {
        int8_t   SVE_tag_addr;
        int8_t   SVE_tag_selector;
        uint8_t  address_format;
        uint8_t  address_length;
        uint8_t  address [20];
    } t_atm_sap_addr;

    struct t_atm_sap_layer2 {
        int8_t   SVE_tag;
        uint8_t  ID_type;
        union  {
            uint8_t  simple_ID;
            uint8_t  user_defined_ID;
        }  ID;
    } t_atm_sap_layer2;

    struct t_atm_sap_layer3 {
        int8_t   SVE_tag;
        uint8_t  ID_type;
        union  {
            uint8_t  simple_ID;
            int32_t  IPI_ID;
            struct  {
                uint8_t  OUI [3];
                uint8_t  PID [2];
            }  SNAP_ID;
            uint8_t  user_defined_ID;
        }  ID;
    } t_atm_sap_layer3;

    struct t_atm_sap_appl {
        int8_t   SVE_tag;
        uint8_t  ID_type;
        union  {
            uint8_t  T_ISO_ID [8];
            struct  {
                uint8_t  OUI [3];
                uint8_t  app_ID [4];
            }  vendor_ID;
            uint8_t  user_defined_ID[8];
        }  ID;
    } t_atm_sap_appl;
}
```

Legal values for the field *t_atm_sap_addr.SVE_tag_addr* are T_ATM_PRESENT and T_ATM_ANY. The semantic meaning of this field is found in section 4.4 of referenced document **ATMNAS**.

Legal values for the field *t_atm_sap_addr.SVE_tag_selector* are T_ATM_PRESENT, T_ATM_ABSENT, and T_ATM_ANY. Note that T_ATM_PRESENT is valid only for ATM Endsystem addresses, and T_ATM_ABSENT is valid only for E.164 addresses. The semantic meaning of this field is found in section 4.4 of referenced document **ATMNAS**.

Legal values for the field *t_atm_sap_addr.address_format* are T_ATM_ENDSYS_ADDR and T_ATM_E164_ADDR. This field is mapped to octet 5 of the Q.2931 ''Called Party Number'' information element.

Legal values for the field *t_atm_sap_addr.address_length* are 0 thru 20. This field is not mapped to any octets of a Q.2931 information element. Instead, it specifies the valid number of array elements in field *t_atm_sap_addr.address*.

Legal values for the field *t_atm_sap_addr.address* can be found in section 5.1.3 of the referenced **UNI** specification (versions 3.0 and 3.1). This field is mapped to octets 6 and beyond of the Q.2931 ''Called Party Number'' information element.

Legal values for the field *t_atm_sap_layer2.SVE_tag* are T_ATM_PRESENT, T_ATM_ABSENT, and T_ATM_ANY. The semantic meaning of this field is found in section 4.4 of referenced document **ATMNAS**.

Legal values for the field *t_atm_sap_layer2.ID_type* are:

| | |
|---|---|
| T_ATM_SIMPLE_ID | identification via ITU encoding |
| T_ATM_USER_ID | identification via a user-defined codepoint. |

This field is not mapped to any octets of a Q.2931 information element. Instead, it specifies the proper union member in the **t_atm_layer2** structure.

Legal values for the field *t_atm_sap_layer2.ID.simple_ID* are:

| | |
|---|---|
| T_ATM_BLLI2_I1745 | I.1745 |
| T_ATM_BLLI2_Q921 | Q.921 |
| T_ATM_BLLI2_X25_LINK | X.25, link layer |
| T_ATM_BLLI2_X25_MLINK | X.25, multilink |
| T_ATM_BLLI2_LAPB | Extended LAPB |
| T_ATM_BLLI2_HDLC_ARM | I.4335, ARM |
| T_ATM_BLLI2_HDLC_NRM | I.4335, NRM |
| T_ATM_BLLI2_HDLC_ABM | I.4335, ABM |
| T_ATM_BLLI2_I8802 | I.8802 |
| T_ATM_BLLI2_X75 | X.75 |
| T_ATM_BLLI2_Q922 | Q.922 |
| T_ATM_BLLI2_I7776 | I.7776 |

This field is mapped to octet 6 (bits 1 thru 5) of the Q.2931 BLLI (''Broadband Low Layer Information'') information element.

Legal values for the field *t_atm_sap_layer2.ID.user_defined_ID* are 0 thru 127. This field is mapped to octet 6a (bits 1 thru 7) of the Q.2931 BLLI information element.

Legal values for the field *t_atm_sap_layer3.SVE_tag* are T_ATM_PRESENT, T_ATM_ABSENT, and T_ATM_ANY. The semantic meaning of this field is found in section 4.4 of referenced document **ATMNAS**.

Legal values for the field *t_atm_sap_layer3.ID_type* are:

| | |
|---|---|
| T_ATM_SIMPLE_ID | identification via ITU encoding |

| | |
|---|---|
| T_ATM_IPI_ID | identification via ISO/IEC TR 9577 (during connection setup) |
| T_ATM_SNAP_ID | identification via SNAP |
| T_ATM_USER_ID | identification via a user-defined codepoint |

This field is not mapped to any octets of a Q.2931 information element. Instead, it specifies the proper union member in the **t_atm_layer3** structure.

Legal values for the field *t_atm_sap_layer3.ID.simple_ID* are:

| | |
|---|---|
| T_ATM_BLLI3_X25 | X.25 |
| T_ATM_BLLI3_I8208 | I.8208 |
| T_ATM_BLLI3_X223 | X.223 |
| T_ATM_BLLI3_I8473 | I.8473 |
| T_ATM_BLLI3_T70 | T.70 |
| T_ATM_BLLI3_I9577 | I.9577 |

Note that a value of T_ATM_BLLI3_I9577 in this field indicates that the identification of the layer 3 protocol is done in the user (data) plane, as specified in ISO/IEC TR 9577. This field is mapped to octet 7 (bits 1 thru 5) of the Q.2931 BLLI information element.

Legal values for the field *t_atm_sap_layer3.ID.IPI_ID* are those values defined by ISO/IEC TR 9577. This field is mapped to octet 7a (bits 1 thru 7) and octet 7b (bit 7) of the Q.2931 BLLI information element.

Legal values for the field *t_atm_sap_layer3.ID.SNAP_ID.OUI* are the 24-bit Organization Unique Identifiers assigned by the IEEE. This field is mapped to octets 8.1 thru 8.3 of the Q.2931 BLLI information element.

Legal values for the field *t_atm_sap_layer3.ID.SNAP_ID.PID* are defined by the organization identified in the preceding field. This field is mapped to octets 8.4 thru 8.5 of the Q.2931 BLLI information element.

Legal values for the field *t_atm_sap_layer3.ID.user_defined_ID* are 0 thru 127. This field is mapped to octet 7a (bits 1 thru 7) of the Q.2931 BLLI information element.

Legal values for the field *t_atm_sap_appl.SVE_tag* are T_ATM_PRESENT, T_ATM_ABSENT, and T_ATM_ANY. The semantic meaning of this field is found in section 4.4 of referenced document **ATMNAS**.

Legal values for the field *t_atm_sap_appl.ID_type* are:

| | |
|---|---|
| T_ATM_ISO_APP_ID | ISO codepoint |
| T_ATM_VENDOR_APP_ID | vendor-specific codepoint |
| T_ATM_USER_APP_ID | identification via a user-defined codepoint |

This field is mapped to octet 5 (bits 1 thru 7) of the Q.2931 BHLI information element. It also specifies the proper union member in the **t_atm_sap_appl** structure.

Legal values for the field *t_atm_sap_appl.ID.T_ISO_ID* are reserved for specification by ISO. At the time of publication, this was an area of further study for ISO. This field is mapped to octets 6 thru 13 of the Q.2931 BHLI information element.

Legal values for the field *t_atm_sap_appl.ID.vendor_ID.OUI* are the 24-bit Organizationally Unique Identifiers assigned by the IEEE. This field is mapped to octets 6 thru 8 of the Q.2931 BHLI information element.

Legal values for the field *t_atm_sap_appl.ID.vendor_ID.app_ID* are specified by the vendor identified in the *vendor_ID.OUI* field. The *vendor_ID.app_ID* field is mapped to octets 9 thru 12 of the Q.2931 BHLI information element.

Legal values for the field *t_atm_sap_appl.ID.user_defined_ID* are 0 through 127. This field is mapped to octets 6 thru 13 of the Q.2931 BHLI information element.

## I.3 Options

Options are formatted according to the structure **t_opthdr** as described in Chapter 13. A compliant ATM transport provider supports all of the options defined in this appendix. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode.

### I.3.1 Signalling-level Options

The protocol level is T_ATM_SIGNALING. For this level, Table I-1 shows the options that are defined.

| Option Name | Type of Option Value | Legal Option Value | Meaning |
|---|---|---|---|
| T_ATM_AAL5 | **struct t_atm_aal5** | see text | ATM adaptation layer 5 |
| T_ATM_TRAFFIC | **struct t_atm_traffic** | see text | data traffic descriptor |
| T_ATM_BEARER_CAP | **struct t_atm_bearer** | see text | ATM service capabilities |
| T_ATM_BHLI | **struct t_atm_bhli** | see text | higher-layer protocol |
| T_ATM_BLLI | **struct t_atm_blli** | see text | lower-layer protocol (1st choice) |
| T_ATM_DEST_ADDR | **struct t_atm_addr** | see text | call responders network address |
| T_ATM_DEST_SUB | **struct t_atm_addr** | see text | call responder's subaddress |
| T_ATM_ORIG_ADDR | **struct t_atm_addr** | see text | call initiators network address |
| T_ATM_ORIG_SUB | **struct t_atm_addr** | see text | call initiator's subaddress |
| T_ATM_CALLER_ID | **struct t_atm_caller_id** | see text | caller's identification attributes |
| T_ATM_CAUSE | **struct t_atm_cause** | see text | cause of disconnection |
| T_ATM_QOS | **struct t_atm_qos** | see text | desired quality of service |
| T_ATM_TRANSIT | **struct t_atm_transit** | see text | public carrier transit network |

**Table I-1** Signaling-level Options

These options are all association-related. See Chapter 13 for the difference between options that are association-related and those that are not.

With the exception of option T_ATM_CAUSE, each of these options may be negotiated for a connection initiated by the transport user. The results of any such negotiation are signalled to the remote device during connection establishment. The negotiation could be done in either of the following ways:

- the *t_connect*( ) function, plus *t_rcvconnect*( ) if needed for asynchronous mode
- the *t_optmgmt*( ) function, if called before *t_connect*( ).

The first of the above two methods is recommended, since some of the option values can change during connection establishment. Use of the first method returns the updated options to the transport user as parameters of the *t_connect*( ) or *t_rcvconnect*( ) function call.

For the case of a transport user passively awaiting incoming calls, only the T_ATM_AAL5 and T_ATM_BLLI options may be negotiated. The results of any such negotiation are signalled to the remote device during connection establishment. The negotiation could be done in either of the following ways:

- the *t_accept*( ) function
- the *t_optmgmt*( ) function on the responding (not listening) transport endpoint, if called before *t_accept*( ).

The first of the above two methods is recommended, since the options proposed by the initiating party are not associated with the responding transport endpoint prior to *t_accept*( ). Thus, the

*t_optmgmt*( ) function call could succeed with invalid options, but the connection would be aborted when *t_accept*( ) is called.

### I.3.2 Absolute Requirements

A request for option T_ATM_AAL5, T_ATM_TRAFFIC, T_ATM_BLLI, or T_ATM_QOS is not an absolute requirement[20]. Any other option request is an absolute requirement.

The above listed options can be negotiated down by the ATM transport provider or peer endsystem if necessary and when appropriate.

### I.3.3 Further Remarks

T_ATM_AAL5

This option is used to signal end-to-end ATM adaptation layer (AAL5) parameters. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.5 of the referenced UNI specification (versions 3.0 and 3.1).

When an incoming connection indication is present, the transport user optionally negotiates this option, which is signalled to the ATM device originating the ATM call. The specific fields within this option that may be modified by the transport user are *forward_max_SDU_size* and *backward_max_SDU_size*. The negotiation could be done in either of the following ways:

- the *t_accept*( ) function
- the *t_optmgmt*( ) function, if called before *t_accept*( ).

The option value consists of a structure **t_atm_aal5** declared as:

```
struct t_atm_aal5 {
    int32_t   forward_max_SDU_size;
    int32_t   backward_max_SDU_size;
    int32_t   SSCS_type;
}
```

Legal values for the field *forward_max_SDU_size* are T_ATM_ABSENT, and 0 through (2**16 – 1). This field is mapped to octets 6.1 and 6.2 of the Q.2931 information element.

Legal values for the field *backward_max_SDU_size* are T_ATM_ABSENT, and 0 thru (2**16 – 1). This field is mapped to octets 7.1 and 7.2 of the Q.2931 information element.

Legal values for the field *SSCS_type* are:

| | |
|---|---|
| T_ATM_ABSENT | no indication |
| T_ATM_NULL | no SSCS layer on top of AAL5 |
| T_ATM_SSCS_SSCOP_REL | SSCOP (assured) SSCS |
| T_ATM_SSCS_SSCOP_UNREL | SSCOP (unassured) SSCS |
| T_ATM_SSCS_FR | frame relay SSCS |

This field is mapped to octet 8.1 of the Q.2931 information element. If, as a default, the transport provider causes this Q.2931 information element field to be present in the connection setup, then this field shall default to a value consistent with the parameter *name*

20. The definition of *absolute requirement* is given in Section 13.3.3 on page 152.

that was specified for *t_open*( ):

| *t_open*( ) name parameter | default SSCS_type |
|---|---|
| AAL5 | T_ATM_NULL |
| SSCOP/AAL5 | T_ATM_SSCS_SSCOP_REL |

Note that if all fields of the option have a value of T_ATM_ABSENT, then this denotes that the entire information element is not present in the Q.2931 network message.

T_ATM_TRAFFIC

This option is used to signal the data bandwidth parameters. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.6 of the referenced **UNI** specification (versions 3.0 and 3.1).

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

```
struct t_atm_traffic_substruct {
    int32_t  PCR_high_priority;
    int32_t  PCR_all_traffic;
    int32_t  SCR_high_priority;
    int32_t  SCR_all_traffic;
    int32_t  MBS_high_priority;
    int32_t  MBS_all_traffic;
    int32_t  tagging;
}

struct t_atm_traffic {
    struct   t_atm_traffic_substruct  forward;
    struct   t_atm_traffic_substruct  backward;
    uint8_t  best_effort;
}
```

Legal values for the field *forward.PCR_high_priority* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 5.1 thru 5.3 of the Q.2931 information element.

Legal values for the field *forward.PCR_all_traffic* are 0 thru (2**24 – 1). This field is mapped to octets 7.1 thru 7.3 of the Q.2931 information element.

Legal values for the field *forward.SCR_high_priority* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 9.1 thru 9.3 of the Q.2931 information element.

Legal values for the field *forward.SCR_all_traffic* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 11.1 thru 11.3 of the Q.2931 information element.

Legal values for the field *forward.MBS_high_priority* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 13.1 thru 13.3 of the Q.2931 information element.

Legal values for the field *forward.MBS_all_traffic* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 15.1 thru 15.3 of the Q.2931 information element.

Legal values for the field *forward.tagging* are T_YES and T_NO. This field is mapped to octet 18.1 (bit 1) of the Q.2931 information element.

Legal values for the field *backward.PCR_high_priority* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 6.1 thru 6.3 of the Q.2931 information element.

Legal values for the field *backward.PCR_all_traffic* are 0 thru (2**24 – 1). This field is mapped to octets 8.1 thru 8.3 of the Q.2931 information element.

Legal values for the field *backward.SCR_high_priority* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 10.1 thru 10.3 of the Q.2931 information element.

Legal values for the field *backward.SCR_all_traffic* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 12.1 thru 12.3 of the Q.2931 information element.

Legal values for the field *backward.MBS_high_priority* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 14.1 thru 14.3 of the Q.2931 information element.

Legal values for the field *backward.MBS_all_traffic* are T_ATM_ABSENT, and 0 thru (2**24 – 1). This field is mapped to octets 16.1 thru 16.3 of the Q.2931 information element.

Legal values for the field *backward.tagging* are T_YES and T_NO. This field is mapped to octet 18.1 (bit 2) of the Q.2931 information element.

Legal values for the field *best_effort* are T_YES and T_NO. This field is mapped to octet 17 of the Q.2931 information element.

T_ATM_BEARER_CAP

This option is used to signal the capabilities of the ATM service. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.7 of the referenced **UNI** specification (versions 3.0 and 3.1).

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

```
struct t_atm_bearer {
    uint8_t  bearer_class;
    uint8_t  traffic_type;
    uint8_t  timing_requirements;
    uint8_t  clipping_susceptibility;
    uint8_t  connection_configuration;
}
```

Legal values for the field *bearer_class* (see ITU Recommendation F.811) are:

| | |
|---|---|
| T_ATM_CLASS_A | bearer class A |
| T_ATM_CLASS_C | bearer class C |
| T_ATM_CLASS_X | bearer class X |

This field is mapped to octet 5 (bits 1 thru 5) of the Q.2931 information element.

Legal values for the field *traffic_type* are:

| | |
|---|---|
| T_ATM_NULL | no indication of traffic type |
| T_ATM_CBR | constant bit rate |
| T_ATM_VBR | variable bit rate |

This field is mapped to octet 5a (bits 3 thru 5) of the Q.2931 information element.

Legal values for the field *timing_requirements* are:

| | |
|---|---|
| T_ATM_NULL | no indication of requirements |
| T_ATM_END_TO_END | end-to-end timing required |
| T_ATM_NO_END_TO_END | end-to-end timing not required |

This field is mapped to octet 5a (bits 1 and 2) of the Q.2931 information element.

Legal values for the field *clipping_susceptibility* are:

| | |
|---|---|
| T_NO | not susceptible to clipping |
| T_YES | susceptible to clipping |

This field is mapped to octet 6 (bits 6 and 7) of the Q.2931 information element.

Legal values for the field *connection_configuration* are:

| | |
|---|---|
| T_ATM_1_TO_1 | point-to-point connection |
| T_ATM_1_TO_MANY | point-to-multipoint connection |

This field is mapped to octet 6 (bits 1 and 2) of the Q.2931 information element.

T_ATM_BHLI

This option is used to signal information about the application that will communicate across the connection. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.8 of the referenced **UNI** specification (versions 3.0 and 3.1).

For the transport user initiating the connection, the option values pertaining to the specification of the remote ATM protocol address are overwritten with the corresponding parameters of the *t_connect*( ) function call. The following fields of this option are set to values found in analogous fields of **t_atm_sap_appl** structure in the destination protocol address when function *t_connect*( ) is invoked:

*ID_type*,
and all members of union
*ID*.

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

```
struct t_atm_bhli {
    int32_t  ID_type;
    union  {
        uint8_t  T_ISO_ID [8];
        struct  {
            uint8_t OUI [3];
            uint8_t app_ID [4];
        } vendor_ID;
        uint8_t  user_defined_ID[8];
    } ID;
}
```

Legal values for the field *ID_type* are:

| | |
|---|---|
| T_ATM_ABSENT | application ID is not indicated |
| T_ATM_ISO_APP_ID | ISO codepoint |
| T_ATM_VENDOR_APP_ID | vendor-specific codepoint |
| T_ATM_USER_APP_ID | identification via a user-defined codepoint |

This field is mapped to octet 5 (bits 1 thru 7) of the Q.2931 information element. The value T_ATM_ABSENT denotes that the entire information element is not present in the Q.2931 network message.

Legal values for the field *ID.T_ISO_ID* are reserved for specification by ISO. At the time of publication, this was an area of further study for ISO. This field is mapped to octets 6 thru 13 of the Q.2931 information element.

Legal values for the field *ID.vendor_ID.OUI* are the 24-bit Organization Unique Identifiers assigned by the IEEE. This field is mapped to octets 6 thru 8 of the Q.2931 information element.

Legal values for the field *ID.vendor_ID.app_ID* are specified by the vendor identified in the *ID.vendor_ID.OUI* field. The *ID.vendor_ID.app_ID* field is mapped to octets 9 thru 12 of the Q.2931 information element.

Legal values for the field *ID.user_defined_ID* are 0 through 127. This field is mapped to octets 6 thru 13 of the Q.2931 BHLI information element.

T_ATM_BLLI

This option is used to signal information about the layer 2 and layer 3 protocols (if any) that will communicate across the connection. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.9 of the referenced **UNI** specification (versions 3.0 and 3.1). Note that this option represents the first choice of a transport user initiating a connection across the ATM network. Up to three such choices can be signalled across the ATM network (however, XTI supports only one choice at this time).

For the transport user initiating the connection, the option values pertaining to the specification of the remote ATM protocol address are overwritten with the corresponding parameters of the *t_connect*( ) function call. The following fields of this option are set to values found in analogous fields of **t_atm_sap_layer2** and **t_atm_sap_layer3** structures in the destination protocol address when function *t_connect*( ) is invoked:

- *layer_2_protocol.ID_type*, and all members of union *layer_2_protocol.ID*
- *layer_3_protocol.ID_type*, and all members of union *layer_3_protocol.ID*.

When an incoming connection indication is present, the transport user optionally negotiates this option, which is signalled to the ATM device originating the ATM call. The specific fields within this option that may be modified by the transport user are:

- *layer_2_protocol.mode, layer_2_protocol.window_size*
- *layer_3_protocol.mode, layer_3_protocol.packet_size*
- *layer_3_protocol.window_size*.

The option negotiation could be done in either of the following ways:

- the *t_accept*( ) function
- the *t_optmgmt*( ) function, if called before *t_accept*( ).

The transport user accepting the incoming connection indication must perform any negotiation according to the guidelines described in section C.3, Annex C of the referenced **UNI** specification, versions 3.0 and 3.1. Support of negotiation described in section C.4 of Annex C is for further study.

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

```
struct t_atm_blli {
    struct  {
        int8_t  ID_type;
        union  {
            uint8_t  simple_ID;
            uint8_t  user_defined_ID;
        }  ID;
        int8_t  mode;
        int8_t  window_size;
    }  layer_2_protocol;
    struct  {
        int8_t  ID_type;
        union  {
            uint8_t  simple_ID;
            int32_t  IPI_ID;
            struct  {
                uint8_t  OUI [3];
                uint8_t  PID [2];
            }  SNAP_ID;
            uint8_t  user_defined_ID;
        }  ID;
        int8_t  mode;
        int8_t  packet_size;
        int8_t  window_size;
    }  layer_3_protocol;
}
```

Legal values for the field *layer_2_protocol.ID_type* are:

| | |
|---|---|
| T_ATM_ABSENT | layer 2 identification is not present |
| T_ATM_SIMPLE_ID | identification via ITU encoding |
| T_ATM_USER_ID | identification via a user-defined codepoint |

This field is not mapped to any octets of a Q.2931 information element. Instead, it specifies the proper union member in the **t_atm_layer2** structure.

Legal values for the field *layer_2_protocol.ID.simple_ID* are:

| | |
|---|---|
| T_ATM_BLLI2_I174 | I.1745 |
| T_ATM_BLLI2_Q921 | Q.921 |
| T_ATM_BLLI2_X25_LINK | X.25, link layer |
| T_ATM_BLLI2_X25_MLINK | X.25, multilink |
| T_ATM_BLLI2_LAPB | Extended LAPB |
| T_ATM_BLLI2_HDLC_ARM | I.4335, ARM |
| T_ATM_BLLI2_HDLC_NRM | I.4335, NRM |
| T_ATM_BLLI2_HDLC_ABM | I.4335, ABM |
| T_ATM_BLLI2_I8802 | I.8802 |
| T_ATM_BLLI2_X75 | X.75 |
| T_ATM_BLLI2_Q922 | Q.922 |
| T_ATM_BLLI2_I7776 | I.7776 |

This field is mapped to octet 6 (bits 1 thru 5) of the Q.2931 information element.

Legal values for the field *layer_2_protocol.ID.user_defined_ID* are 0 thru 127. This field is mapped to octet 6a (bits 1 thru 7) of the Q.2931 BLLI information element.

Legal values for the field *layer_2_protocol.mode* are T_ATM_ABSENT, T_ATM_BLLI_NORMAL_MODE, and T_ATM_BLLI_EXTENDED_MODE. This field is mapped to octet 6a (bits 6 and 7) of the Q.2931 information element.

Legal values for the field *layer_2_protocol.window_size* are T_ATM_ABSENT, and 1 thru 127. This field is mapped to octet 6b (bits 1 thru 7) of the Q.2931 information element.

Legal values for the field *layer_3_protocol.ID_type* are:

| | |
|---|---|
| T_ATM_ABSENT | no layer 3 protocol identification |
| T_ATM_SIMPLE | ID identification via ITU encoding |
| T_ATM_IPI_ID | identification via ISO / IEC TR 9577 |
| T_ATM_SNAP_ID | identification via SNAP |
| T_ATM_USER_ID | identification via a user-defined codepoint |

This field is not mapped to any octets of the Q.2931 information element. Instead, it specifies the proper union member in structure **t_atm_blli**.

Legal values for the field *layer_3_protocol.ID.simple_ID* are:

| | |
|---|---|
| T_ATM_BLLI3_X25 | X.25 |
| T_ATM_BLLI3_I8208 | I.8208 |
| T_ATM_BLLI3_X223 | X.223 |
| T_ATM_BLLI3_I8473 | I.8473 |
| T_ATM_BLLI3_T70 | 70 |
| T_ATM_BLLI3_I9577 | I.9577 (during connection setup) |

Note that a value of T_ATM_BLLI3_I9577 in this field indicates that the identification of the layer 3 protocol is done in the user (data) plane, as specified in ISO / IEC TR 9577. This field is mapped to octet 7 (bits 1 thru 5) of the Q.2931 information element.

Legal values for the field *layer_3_protocol.ID.IPI_ID* are T_ATM_ABSENT, and those values defined by ISO / IEC TR 9577. This field is mapped to octet 7a (bits 1 thru 7) and octet 7b (bit 7) of the Q.2931 information element. Note that the value T_ATM_ABSENT is used to signal that the identification of the network-layer protocol is carried with each TSDU in the data plane, according to codepoints defined by ISO / IEC TR 9577.

Legal values for the field *layer_3_protocol.ID.SNAP_ID.OUI* are the 24-bit Organization Unique Identifiers assigned by IEEE. This field is mapped to octets 8.1 thru 8.3 of the Q.2931 information element.

Legal values for the field *layer_3_protocol.ID.SNAP_ID.PID* are defined by the organization identified in the preceding field. This field is mapped to octets 8.4 thru 8.5 of the Q.2931 information element.

Legal values for the field *layer_3_protocol.ID.user_defined_ID* are 0 thru 127. This field is mapped to octet 7a (bits 1 thru 7) of the Q.2931 BLLI information element.

Legal values for the field *layer_3_protocol.mode* are T_ATM_ABSENT, T_ATM_BLLI_NORMAL_MODE, and T_ATM_BLLI_EXTENDED_MODE. This field is mapped to octet 7a (bits 6 and 7) of the Q.2931 information element.

Legal values for the field *layer_3_protocol.packet_size* are:

T_ATM_ABSENT
T_ATM_PACKET_SIZE_16
T_ATM_PACKET_SIZE_32
T_ATM_PACKET_SIZE_64
T_ATM_PACKET_SIZE_128

T_ATM_PACKET_SIZE_256
T_ATM_PACKET_SIZE_512
T_ATM_PACKET_SIZE_1024
T_ATM_PACKET_SIZE_2048
T_ATM_PACKET_SIZE_4096

This field is mapped to octet 7b (bits 1 thru 4) of the Q.2931 information element.

Legal values for the field *layer_3_protocol.window_size* are T_ATM_ABSENT, and 1 thru 127. This field is mapped to octet 7c (bits 1 thru 7) of the Q.2931 information element.

T_ATM_DEST_ADDR

This option is used to signal the ATM network address of the connections destination. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.11 of the referenced **UNI** specification (versions 3.0 and 3.1).

For the transport user initiating the connection, the option values pertaining to the specification of the remote ATM protocol address are overwritten with the corresponding parameters of the *t_connect*( ) function call. The following fields of this option are set to values found in analogous fields of the destination protocol address when function *t_connect*( ) is invoked:

- *address_format*
- *address_length*
- *address*.

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

```
struct t_atm_addr {
    int8_t   address_format;
    uint8_t  address_length;
    uint8_t  address [20];
}
```

Legal values for the field *address_format* are T_ATM_ENDSYS_ADDR and T_ATM_E164_ADDR. This field is mapped to octet 5 of the Q.2931 information element.

Legal values for the field *address_length* are 0 thru 20. This field is not mapped to any octets of a Q.2931 information element. Instead, it specifies the valid number of array elements in field *address*.

Legal values for the field *address* can be found in section 5.1.3 of the referenced **UNI** specification (versions 3.0 and 3.1). This field is mapped to octets 6 and beyond of the Q.2931 information element.

T_ATM_DEST_SUB

This option is used to signal the ATM subaddress of the connections destination. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.12 of the referenced **UNI** specification (versions 3.0 and 3.1).

The option value consists of a structure **t_atm_addr** (see option T_ATM_DEST_ADDR for the structure's declaration). Note that for this option, field *address_format* must have a value of either T_ATM_NSAP_ADDR or T_ATM_ABSENT.

A value of T_ATM_ABSENT in field *address_format* of structure **t_atm_addr** indicates that the information element is not present in the Q.2931 signalling message.

T_ATM_ORIG_ADDR

This option is used to signal the ATM network address of the connections originator. The

description of the information element that is signalled across the ATM network can be found in section 5.4.5.13 of the referenced **UNI** specification (versions 3.0 and 3.1).

The option value consists of a structure **t_atm_addr** (see option T_ATM_DEST_ADDR for the structure's declaration).

A value of T_ATM_ABSENT in field *address_format* of structure **t_atm_addr** indicates that the information element is not present in the Q.2931 signalling message.

T_ATM_ORIG_SUB

This option is used to signal the ATM subaddress of the connections originator. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.14 of the referenced **UNI** specification (versions 3.0 and 3.1).

The option value consists of a structure **t_atm_addr** (see option T_ATM_DEST_ADDR for the structure's declaration). Note that for this option, field *address_format* must have a value of either T_ATM_NSAP_ADDR or T_ATM_ABSENT.

A value of T_ATM_ABSENT in field *address_format* of structure **t_atm_addr** indicates that the information element is not present in the Q.2931 signalling message.

T_ATM_CALLER_ID

This option is used to signal additional attributes concerning the network address of the connection's originator. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.13 of the referenced **UNI** specification (versions 3.0 and 3.1).

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

```
struct t_atm_caller_id {
    int8_t   presentation;
    uint8_t  screening;
}
```

Legal values for the field *presentation* are:

T_ATM_ABSENT
T_ATM_PRES_ALLOWED
T_ATM_PRES_RESTRICTED
T_ATM_PRES_UNAVAILABLE

This field is mapped to octet 5a (bits 6 and 7) of the Q.2931 information element.

Legal values for the field *screening* are:

T_ATM_ABSENT
T_ATM_USER_ID_NOT_SCREENED
T_ATM_USER_ID_PASSED_SCREEN
T_ATM_USER_ID_FAILED_SCREEN
T_ATM_NETWORK_PROVIDED_ID

This field is mapped to octet 5a (bits 1 and 2) of the Q.2931 information element.

T_ATM_CAUSE

This option is used to signal the cause of a disconnection. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.15 of the referenced **UNI** specification (versions 3.0 and 3.1).

Upon disconnection, the transport user optionally negotiates this option, the results of which are signalled to the remote ATM device. Any such negotiation must be performed

via the *t_optmgmt*( ) function.

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

```
struct t_atm_cause {
    int8_t   coding_standard;
    uint8_r  location;
    uint8_r  cause_value;
    uint8_r  diagnostics [4];
}
```

Legal values for the field *coding_standard* are T_ATM_ABSENT, T_ATM_ITU_CODING, and T_ATM_NETWORK_CODING. This field is mapped to octet 2 (bits 6 and 7) of the Q.2931 information element. The value of T_ATM_ABSENT denotes that the entire information element is not present in the Q.2931 network message.

Legal values for the field *location* are:

T_ATM_LOC_USER
T_ATM_LOC_LOCAL_PRIVATE_NET
T_ATM_LOC_LOCAL_PUBLIC_NET
T_ATM_LOC_TRANSIT_NET
T_ATM_LOC_REMOTE_PUBLIC_NET
T_ATM_LOC_REMOTE_PRIVATE_NET
T_ATM_LOC_INTERNATIONAL_NET
T_ATM_LOC_BEYOND_INTERWORKING

This field is mapped to octet 5 (bits 1 thru 4) of the Q.2931 information element.

Legal values for the field *cause_value* are listed in a full-page table in the referenced **UNI** specification. This field is mapped to octet 6 (bits 1 thru 7) of the Q.2931 information element.

Legal values for the field *diagnostics* are beyond the scope of this specification. This field is mapped to octets 7 and beyond of the Q.2931 information element.

T_ATM_QOS

This option is used to signal the desired quality of service. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.18 of the referenced **UNI** specification (versions 3.0 and 3.1).

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

```
struct t_atm_qos_substruct {
    int32_t  coding_standard;
}

struct t_atm_qos {
    int8_t   coding_standard;
    struct t_atm_qos_substruct  forward;
    struct t_atm_qos_substruct  backward;
}
```

Legal values for the field *coding_standard* are T_ATM_ABSENT, T_ATM_ITU_CODING, and T_ATM_NETWORK_CODING. This field is mapped to octet 2 (bits 6 and 7) of the Q.2931 information element. The value of T_ATM_ABSENT denotes that the entire information element is not present in the Q.2931 network message.

Legal values for the field *forward.qos_class* are:

T_ATM_QOS_CLASS_0
T_ATM_QOS_CLASS_1
T_ATM_QOS_CLASS_2
T_ATM_QOS_CLASS_3
T_ATM_QOS_CLASS_4

This field is mapped to octet 5 of the Q.2931 information element.

Legal values for the field *backward.qos_class* are the same as those specified for field *forward.qos_class* above. This field ( *backward.qos_class*) is mapped to octet 6 of the Q.2931 information element.

T_ATM_TRANSIT

This option is used to signal the selection of the inter-exchange public carrier. The description of the information element that is signalled across the ATM network can be found in section 5.4.5.22 of the referenced **UNI** specification (versions 3.0 and 3.1).

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

```
struct t_atm_transit  {
    uint8_t  length;
    uint8_t  network_id[];
}
```

The field *length* specifies how many characters in array *network_id* are valid.

Legal values for the field *length* are 0 thru 255. The value of 0 denotes that the entire information element is not present in the Q.2931 network message.

Legal values for the field *network_id* are beyond the scope of this document. This field is mapped to octets 6 and beyond of the Q.2931 information element.

## I.4 Existing Functions

*t_accept*( )

The parameter *call→addr* is ignored by the ATM transport provider.

The parameter *call→opt* is used only to set negotiable ATM connection attributes. These attributes (and XTI options) are T_ATM_AAL5 and T_ATM_BLLI.

Note that the transport provider must queue data sent via *t_snd*( ) until the end-to-end data path is operational.

*t_bind*( )

When a transport user wishes to passively wait for incoming connect indications through a transport endpoint, then *t_bind*( ) is called with parameter *req→qlen* having a non-zero value. For ATM, the following additional restrictions apply:

- The ATM protocol address bound to a transport endpoint must conform to the ATM Forum guidelines (referenced document **ATMNAS**) for the specification of a SAP address. The SAP address is a vector that includes fields for the ATM network address (with selector byte), identification of a layer-2 protocol, identification of a layer-3 protocol, and identification of an application.
- The ATM Forum guidelines (referenced document **ATMNAS**) determines uniqueness of an ATM protocol address. If *t_bind*( ) is called with a protocol address considered equivalent to an existing bound address, then [TADDRBUSY] is returned.

  An implementation may optionally allow at most one application to bind to the ‘‘wildcard catch-all’’ SAP of (T_ATM_ANY, T_ATM_ANY, T_ATM_ANY, T_ATM_ANY, T_ATM_ANY) for the 5 SAP vector elements.
- ATM transport providers must support a maximum of one outstanding connect indication; therefore, parameter *ret→qlen* must contain a value of 1 to reflect the downward negotiation. The transport provider must queue incoming calls that have not been indicated to the transport user, up to some maximum queue size that is implementation dependent.

When a transport user wishes to initiate outgoing connect requests through a transport endpoint, it is recommended that *t_bind*( ) be called with parameters *req* and *ret* being null pointers.

If *t_bind*( ) is called with parameter *req→addr.buf* pointing to an ATM network or protocol address, and parameter *req→qlen* having a value of zero; then the ATM provider must perform the following:

- If *t_bind*( ) is called with a non-null pointer for parameter *ret*, then upon return, parameter *ret→addr.buf* contains the bound address.
- The transport provider saves the bound address for future use in case *t_getprotaddr*( ) is called.
- The bound address is overwritten with the destination protocol SAP address if *t_accept*( ) is called and this transport endpoint becomes the endpoint upon which the connection is established.
- The bound address is not passed to the peer entity when making an outbound connection.

*t_close*( )

If the transport user wishes to convey the cause of the disconnection to the remote user, then option T_ATM_CAUSE must be negotiated via *t_optmgmt*( ) prior to calling *t_close*( ).

*t_connect*( )

Parameter *sndcall→addr* specifies the ATM protocol address of the destination transport user. This ATM protocol address must conform to the ATM Forum guidelines (see referenced document **ATMNAS**) for the specification of a SAP address. The SAP address is a vector that includes fields for the ATM network address (with selector byte), identification of a layer-2 protocol, identification of a layer-3 protocol, and identification of an application.

*t_getinfo*( )

The following information parameters are returned:

| Parameters | Before Call | After Call | |
|---|---|---|---|
| | | AAL-5 | SSCOP / AAL-5 |
| *fd* | x | / | / |
| *info->addr* | / | x | x |
| *info->options* | / | x | x |
| *info->tsdu* | / | $1 \leq x \leq 65{,}536$ | $1 \leq x \leq 65{,}528$ |
| *info->etsdu* | / | –2 | –2 |
| *info->connect* | / | –2 | –2 |
| *info->discon* | / | –2 | –2 |
| *info->servtype* | / | T_COTS | T_COTS |
| *info->flags* | / | 0 | 0 |

*t_listen*( )

The parameter *call→addr* is not the ATM protocol address of the calling transport user; it is merely the network address of the calling user's ATM device. This address may not be sufficient to be the destination protocol address in the *t_connect*( ) function.

Upon successful return of *t_listen*( ), the options contained in parameter *call* are those received in the Q.2931 signalling SETUP message from the ATM network.

*t_look*( )

A new event, T_LEAFCHANGE, is defined which is used by event management to notify a transport user when the status of a leaf has changed on a point-to-multipoint connection.

*t_open*( )

The choice of whether or not the transport provider is requested to implement the SSCOP protocol above AAL-5 in the data plane is conveyed via parameter name. The SSCOP protocol provides a reliable data delivery service. See the notes on function *t_getinfo*( ) in this section for a discussion of parameter *info*.

*t_rcvdis*( )

The parameter reason is an 8-bit cause value that is sent across the ATM network in octet 6 of the Q.2931 Cause information element. Additional cause information may be obtained from the T_ATM_CAUSE option via *t_optmgmt*( ).

*t_snd*( )

ATM does not support the transport of expedited data. If the transport user sets the T_EXPEDITED flag in parameter flags, then error [TBADFLAG] is returned.

ATM does not support a zero-length TSDU. If parameter *nbytes* is zero, then [TBADDATA] is returned if either:

- the T_MORE flag (in parameter *flags*) is set

- the T_MORE flag is not set and this is the first *t_snd*( ) call in state T_DATAXFER
- the T_MORE flag is not set and the preceding *t_snd*( ) call completed a TSDU (T_MORE flag was not set in preceding call).

When the connection present at the transport endpoint is a leaf on a point-to-multipoint connection, the transport provider returns [TNOTSUPPORT] for *t_snd*( ).

*t_snddis*( )

If the transport user wishes to convey the cause of the disconnection to the remote user, then option T_ATM_CAUSE must be negotiated via *t_optmgmt*( ) prior to calling *t_snddis*( ).

## I.5 Implementation Notes

This section maps the functions of XTI onto the primitives specified in referenced document **ATMNAS**). This mapping information is provided as guidance for the design and development of ATM Transport Providers.

*t_accept*( )

This function implements the ATM Forums *ATM_accept_incoming_call* primitive. If any options are present in parameter *call→opt*, then each of these options is an implementation of the ATM Forum's *ATM_set_connection_attributes* primitive.

The return of this function can be mapped to the following ATM Forum primitives:

- If successful for the setup of a point-to-point connection, the function's return implements the ATM Forums *ATM_P2P_call_active* primitive.
- If successful for the setup of a point-to-multipoint connection, the function's return implements the ATM Forums *ATM_P2MP_call_active* primitive.

*t_bind*( )

For transport users passively waiting for incoming connect indications, this function implements the ATM Forum primitives *ATM_prepare_incoming_call* and *ATM_wait_on_incoming_call*. For transport users initiating connect requests, this function implements the ATM Forums *ATM_prepare_outgoing_call* primitive.

Note that parameter *queue_size* of the ATM Forum's *ATM_prepare_incoming_call* primitive is different to XTI's *qlen*: The parameter *queue_size* is the maximum number of incoming calls that have arrived but have not yet been presented to the transport user, while the parameter *qlen* is the maximum number of incoming calls that have been presented to the transport user but not yet accepted. Therefore, *queue_size* cannot be set via the *t_bind*( ) function; the value is determined by the transport provider..

*t_close*( )

Normally, this function is called from the T_UNBND state; in this case, there is no needed mapping to the ATM Forum primitives. XTI also allows this function to be legally called from other states as well. When that occurs, this function implements the ATM Forum's *ATM_abort_connection* primitive. However, the transport user cannot specify the cause of the aborted connection.

*t_connect*( )

The invocation of this function implements the ATM Forum's *ATM_connect_outgoing_call* primitive. If any options are present in parameter *sndcall→opt*, then each of these options is an implementation of the ATM Forum's *ATM_set_connection_attributes* primitive. If any options are present in parameter *rcvcall→optc*, then each of these options is an implementation of the ATM Forum's *ATM_query_connection_attributes* primitive.

Additionally, when the transport endpoint is in synchronous mode, the return of this function can be mapped to the following ATM Forum primitives:

- If successful for the setup of a point-to-point connection, the function's return implements the ATM Forums *ATM_P2P_call_active* primitive.
- If successful for the setup of a point-to-multipoint connection, the function's return implements the ATM Forums *ATM_P2MP_call_active* primitive.
- If unsuccessful for the setup of a connection, the function's return has no needed mapping to an ATM Forum primitive.

*t_listen*( )
The successful return of this function is an implementation of the ATM Forum's *ATM_arrival_of_incoming_call* primitive.

The ATM network address (of the calling user) returned in parameter *call→addr* is an implementation of the ATM Forum's *ATM_query_connection_attributes* primitive, where the connection attribute (and XTI option) being queried is T_ATM_ORIG_ADDR. If any options are present in parameter *call→opt*, then each of these options is an implementation of the ATM Forum's *ATM_query_connection_attributes* primitive.

*t_open*( )
This function implements the ATM Forums *ATM_associate_endpoint* primitive.

*t_optmgmt*( )
For any option values that the transport user attempts to negotiate (present in parameter *req→opt* when parameter *req→flags* equals T_NEGOTIATE), each of these options is an implementation of the ATM Forum's *ATM_set_connection_attributes* primitive. For any options present in parameter *ret→opt*, each of these options is an implementation of the ATM Forum's *ATM_query_connection_attributes* primitive.

*t_rcv*( )
The invocation of this function implements the ATM Forum's *ATM_receive_data* primitive. Note that the ATM Forum's primitive can be further classified as either a ''polling implementation'' or a ''blocking implementation''. XTI's synchronous mode corresponds to ''blocking implementation''; asynchronous mode corresponds to ''polling implementation''.

*t_rcvconnect*( )
The return of this function can be mapped to the following ATM Forum primitives:

- If successful for the setup of a point-to-point connection, the function's return implements the ATM Forum's *ATM_P2P_call_active* primitive.
- If successful for the setup of a point-to-multipoint connection, the function's return implements the ATM Forum's *ATM_P2MP_call_active* primitive.
- If unsuccessful for the setup of a connection, the function's return has no needed mapping to an ATM Forum primitive.

If any options are present in parameter *call→opt*, then each of these options is an implementation of the ATM Forum's *ATM_query_connection_attributes* primitive.

*t_rcvdis*( )
The successful return of this function implements the ATM Forums *ATM_call_release* (indication) primitive.

*t_snd*( )
The invocation of this function implements the ATM Forums *ATM_send_data* primitive.

*t_snddis*( )
Depending on the XTI state, invocation of this function implements the following ATM Forum primitives:

| | |
|---|---|
| state T_DATAXFER: | *ATM_call_release* (request) or *ATM_abort_connection* |
| state T_INCON: | *ATM_reject_incoming_call* |
| all other valid states: | *ATM_abort_connection*. |

*t_unbind*( )
There is no ATM Forum primitive to which this function can be mapped. From the

perspective of the ATM Forum state machine, this function does the following:

- moves any future outgoing connections on this endpoint from state A2 to A1
- moves any future incoming connections on this endpoint from state A4 to A1.

*t_addleaf*( )
This function implements the ATM Forum's *ATM_add_party* primitive. If successful in the addition of the leaf, the function's return implements the ATM Forum's *ATM_add_party_success* primitive.

*t_removeleaf*( )
This function implements the ATM Forum's *ATM_drop_party* (Request) primitive.

*t_rcvleafchange*( )
This function implements the following ATM Forum primitives:

*ATM_add_party_success*
*ATM_add_party_reject*
*ATM_drop_party*(Indication)

## I.6 New Functions

The following new functions are defined in the following man-pages:

*t_addleaf*( )
*t_removeleaf*( )
*t_rcvleafchange*( )

**NAME**

t_addleaf - add a leaf to a point-to-multipoint connection

**SYNOPSIS**

```
#include <xti.h>

int32_t t_addleaf(fd, leafid, addr, int32_t fd, int32_t leafid,
    struct netbuf *addr);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *leafid* | x | / |
| *addr.maxlen* | / | / |
| *addr.len* | x | / |
| *addr.buf* | x (x) | / |

This function enables a transport user to add a leaf to a point-to-multipoint connection. This function can only be issued in the T_DATAXFER state. The parameter *fd* identifies the local transport endpoint that serves as the root of the point-to-multipoint connection. The parameter *leafid*, provided by the transport user, will be used by subsequent functions *t_removeleaf*( ) and *t_rcvleafchange*( ) to identify the particular leaf being added. The parameter *addr* is the address of the device being added as a leaf.

The values used for *leafid* may be used simultaneously on point-to-multipoint connections other than the connection indicated by *fd*.

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

**VALID STATES**

T_DATAXFER

**ERRORS**

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

[TBADF]
The specified file descriptor does not refer to a transport endpoint. Also used when the connection is not point-to-multipoint.

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

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

[TADDRBUSY]
This transport provider does not support more than one instance of a particular leaf on a given point-to-multipoint connection. This error indicates that the leaf is already a participant in the point-to-multipoint connection.

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

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

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

[TLOOK]
An asynchronous event, which requires attention, has occurred.

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

**RETURN VALUE**

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

**SEE ALSO**

*t_removeleaf*( ), *t_rcvleafchange*( ).

**ATM PROTOCOL SPECIFICS**

The parameter *addr* is filled with a **t_atm_addr** structure and represents the ATM network address of the leaf being added.

**NAME**

t_removeleaf - drop a leaf from a point-to-multipoint connection

**SYNOPSIS**

```
#include <xti.h>

int32_t t_removeleaf (fd, leafid, reason, int32_t fd, int32_t leafid,
    int32_t reason);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *leafid* | x | / |
| *reason* | x | / |

This function is used to initiate an abortive removal of a leaf from an already established point-to-multipoint connection. This function can only be issued in the T_DATAXFER state. The parameter *fd* identifies the local connection endpoint that serves as the root of the point-to-multipoint connection, and parameter *leafid* identifies the leaf that is being removed from the connection. The parameter *reason* specifies the reason for the leaf removal through a protocol-dependent reason code.

**VALID STATES**

T_DATAXFER

**ERRORS**

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

[TBADF]
The specified file descriptor does not refer to a transport endpoint. Also used when the connection is not point-to-multipoint.

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

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

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

[TLOOK]
An asynchronous event, which requires attention, has occurred.

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

**RETURN VALUE**

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

**SEE ALSO**

*t_addleaf*( ), *t_rcvleafchange*( ).

**CAVEATS**

*t_removeleaf*( ) is an abortive removal of the leaf. Therefore, *t_removeleaf*( ) may cause data previously sent via *t_snd*( ) to not be received by the leaf (even if an error is returned).

**ATM PROTOCOL SPECIFICS**

The parameter *reason* is an 8-bit cause value that is sent across the ATM network in octet 6 of the Q.2931 Cause information element.

**NAME**

t_rcvleafchange - receive an indication about a leaf in a point-to-multipoint connection

**SYNOPSIS**

```
#include <xti.h>

int32_t t_rcvleafchange (fd, change, int32_t fd,
    struct t_leaf_status *change);
```

**DESCRIPTION**

| Parameters | Before call | After call |
|---|---|---|
| *fd* | x | / |
| *change.leafid* | / | x |
| *change.status* | / | x |
| *change.reason* | / | ? |

This function is used to determine the status change of a leaf on a point-to-multipoint connection. This function can only be issued in the T_DATAXFER state. The parameter *fd* identifies the local connection endpoint that serves as the root of the point-to-multipoint connection, and parameter *change* points to a **t_leaf_status** structure containing the following members:

```
int32_t  leafid;
int32_t  status;
int32_t  reason;
```

The field *leafid* identifies the leaf whose status has changed, and field *status* specifies the change (either T_CONNECT or T_DISCONNECT). When *status* has a value of T_CONNECT, field *reason* is meaningless. When *status* has a value of T_DISCONNECT, field *reason* specifies the reason why the leaf was removed from the point-to-multipoint connection or why a pending addleaf failed, through a protocol-dependent reason code.

**VALID STATES**

T_DATAXFER

**ERRORS**

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

[TBADF]
The specified file descriptor does not refer to a transport endpoint. Also used when the connection is not point-to-multipoint.

[TNODATA]
No leaf change indication currently exists on the specified point-to-multipoint connection.

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

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

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

[TLOOK]
An asynchronous event, which requires attention, has occurred.

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

**RETURN VALUE**

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

**SEE ALSO**

*t_addleaf*( ), *t_removeleaf*( ).

**ATM PROTOCOL SPECIFICS**

The parameter *reason* is an 8-bit cause value that is sent across the ATM network in octet 6 of the Q.2931 Cause information element.

*Appendix J*

# *ATM Transport Protocol Information for Sockets*

## J.1 General

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

The following general notes apply:

- This version of Sockets supports a subset of the functions specified by the ATM Forum as the User-Network Interface (see referenced document **UNI**), versions 3.0 and 3.1.
- Frequent reference is made to the ATM Forum's ''Native ATM Services: Semantic Description, Version 1.0'', identified as referenced document **ATMNAS**.
- ATM supports connection-oriented sockets only. Specifically, the socket type is SOCK_SEQPACKET. At the current time, only AAL-5 message mode is supported. Each sockets message is carried across the network in the payload field of a single AAL-5 PDU. AAL1 and User-defined AAL are not supported.
- ATM supports both reliable and unreliable data transport services. Reliable service is obtained by specifying ATM_PROTO_SSCOP as the protocol when calling *socket*( ). Unreliable service is obtained by specifying ATM_PROTO_AAL5 as the protocol when calling *socket*( ).
- ATM addresses (both network and protocol) are defined in Appendix I( ATM Transport Protocol Information for XTI) of this specification.
- When a transport user passively waits for incoming connect indications, the ATM protocol address bound to the socket must conform to the ATM Forum guidelines (see referenced document **ATMNAS**) for the specification of a SAP address. The SAP address is a vector that includes fields for the ATM network address (with selector byte), identification of a layer-2 protocol, identification of a layer-3 protocol, and identification of an application
- AAL and BLLI negotiation is not supported since *accept*( ) contains no provision for examining and modifying connection attributes.
- The ATM transport provider does not support ATM PVCs.
- The ATM transport provider does not support a mechanism to specify the congestion indication bit and the user-user byte in AAL5.

## J.2 Existing Functions

*accept*( )

The parameter *address*, if non-null, points to the ATM network address of the peer connecting socket. The ATM protocol address of the peer connecting socket is unavailable.

Note that the transport protocol must queue data sent via *send*( ), *sendmsg*( ), or *sendto*( ) until the end-to-end data path is operational.

*bind*( )

When a transport user wishes to passively wait for incoming connections through a socket, the following additional restrictions apply for ATM transports:

- The ATM protocol address bound to a socket must conform to the ATM Forum guidelines (referenced document **ATMNAS**) for the specification of a SAP address. The SAP address is a vector that includes fields for the ATM network address (with selector byte), identification of a layer-2 protocol, identification of a layer-3 protocol, and identification of an application.
- The ATM Forum guidelines (referenced document **ATMNAS**) determines uniqueness of an ATM protocol address. If *bind*( ) is called with a protocol address considered equivalent to an existing bound address, then [EADDRINUSE] is returned.

When a transport user wishes to initiate outgoing connections through a socket, then *bind*( ) should not be called.

*close*( )

If the transport user wishes to convey the cause of the disconnection to the peer socket, then socket option T_ATM_CAUSE must be set before function *close*( ) is invoked.

*connect*( )

Parameter *address* specifies the ATM protocol address of the destination transport user. This ATM protocol address must conform to the ATM Forum guidelines (referenced document **ATMNAS**) for the specification of a SAP address. The SAP address is a vector that includes fields for the ATM network address (with selector byte), identification of a layer-2 protocol, identification of a layer-3 protocol, and identification of an application.

*getpeername*( )

The parameter *address*, if meaningful, points to one of the following:

- an ATM network address, for sockets in which the transport user passively waits for an incoming connection
- an ATM protocol address, for sockets in which the transport user actively initiates an outgoing connection.

*getsockname*( )

The parameter *address*, if meaningful, points to one of the following:

- an ATM protocol address, for sockets in which the transport user passively waits for an incoming connection
- an ATM network address, for sockets in which the transport user actively initiates an outgoing connection.

*getsockopt*( )

ATM transport protocol options use T_ATM_SIGNALING for the parameter level. The ATM-specific options are listed in the table below. Most of these options are defined in Appendix I in this specification. Option T_ATM_LEAF_IND is defined in Section J.3 on page 431.

| Option Name | Type of Option Value | Meaning |
|---|---|---|
| T_ATM_AAL5 | struct t_atm_aal5 | ATM adaptation layer 5 |
| T_ATM_TRAFFIC | struct t_atm_traffic | data traffic descriptor |
| T_ATM_BEARER_CAP | struct t_atm_bearer | ATM service capabilities |
| T_ATM_BHLI | struct t_atm_bhli | higher-layer protocol |
| T_ATM_BLLI | struct t_atm_blli | lower-layer protocol (1st choice) |
| T_ATM_DEST_ADDR | struct t_atm_addr | call responders network address |
| T_ATM_DEST_SUB | struct t_atm_addr | call responder's subaddress |
| T_ATM_ORIG_ADDR | struct t_atm_addr | call initiators network address |
| T_ATM_ORIG_SUB | struct t_atm_addr | call initiator's subaddress |
| T_ATM_CALLER_ID | struct t_atm_caller_id | caller's identification attributes |
| T_ATM_CAUSE | struct t_atm_cause | cause of disconnection |
| T_ATM_QOS | struct t_atm_qos | desired quality of service |
| T_ATM_TRANSIT | struct t_atm_transit | public carrier transit network |
| T_ATM_LEAF_IND | struct t_atm_leaf_ind | indication of leaf status change |

*recv*( )

The parameter *flags* must equal a value of 0 or MSG_PEEK.

*recvfrom*( )

Note that ATM protocol supports only connection-oriented sockets.

*recvmsg*( )

Note that ATM protocol supports only connection-oriented sockets.

*send*( )

The parameter *flags* must equal a value of 0.

When the connection present at the socket is a leaf on a point-to-multipoint connection, any invocation of function *send*( )
results in [EOPNOTSUPP].

*sendmsg*( )

Note that ATM protocol supports only connection-oriented sockets.

*sendto*( )

Note that ATM protocol supports only connection-oriented sockets.

*setsockopt*( )

ATM transport protocol options use SOL_ATM_SIGNALING for the parameter level. The ATM-specific options are listed in the table below. Most of these options are defined in Appendix I. Options T_ATM_ADD_LEAF and T_ATM_DROP_LEAF are defined in Section J.3 on page 431.

| Option Name | Type of Option Value | Meaning |
|---|---|---|
| T_ATM_AAL5 | struct t_atm_aal5 | ATM adaptation layer 5 |
| T_ATM_TRAFFIC | struct t_atm_traffic | data traffic descriptor |
| T_ATM_BEARER_CAP | struct t_atm_bearer | ATM service capabilities |
| T_ATM_BHLI | struct t_atm_bhli | higher-layer protocol |
| T_ATM_BLLI | struct t_atm_blli | lower-layer protocol (1st choice) |
| T_ATM_DEST_ADDR | struct t_atm_addr | call responders network address |
| T_ATM_DEST_SUB | struct t_atm_addr | call responder's subaddress |
| T_ATM_ORIG_ADDR | struct t_atm_addr | call initiators network address |
| T_ATM_ORIG_SUB | struct t_atm_addr | call initiator's subaddress |
| T_ATM_CALLER_ID | struct t_atm_caller_id | caller's identification attributes |
| T_ATM_CAUSE | struct t_atm_cause | cause of disconnection |
| T_ATM_QOS | struct t_atm_qos | desired quality of service |
| T_ATM_TRANSIT | struct t_atm_transit | public carrier transit network |
| T_ATM_ADD_LEAF | struct t_atm_add_leaf | add a leaf to a connection |
| T_ATM_DROP_LEAF | struct t_atm_drop_leaf | remove a leaf from a connection |

*shutdown*( )

This function has a local scope only; no control information is transferred across the ATM network.

*socket*( )

The following parameter values should be used for ATM protocol transports:

- *domain* = AF_ATM
- *type* = SOCK_SEQPACKET
- *protocol* = ATM_PROTO_AAL5 or ATM_PROTO_SSCOP

If parameter *protocol* signifies use of the SSCOP protocol, it is recommended that the transport provider use the default of T_ATM_SSCS_SSCOP_REL in field *SSCS_type* of option T_ATM_AAL5.

*socketpair*( )

This function is not supported by an ATM domain. If the function is invoked, the transport provider returns [EOPNOTSUPP].

## J.3 Point-to-Multipoint Connections

To support point-to-multipoint connections, procedures are defined for the addition and removal of other parties (called *leafs* in this specification) to the connection. These procedures are performed only by the root of the connection. This section specifies how socket options are used within X/Sockets to accomplish these functions. For more information, see section 4.2 (''SVC Provisioning'') of referenced document **ATMNAS**.

### J.3.1 Adding a Leaf

T_ATM_ADD_LEAF

This option is used to request that a leaf be added to an existing point-to-multipoint connection. Note that this option is ''write-only''; it is a valid parameter of *setsockopt*( ), but not *getsockopt*( ).

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

```
struct t_atm_add_leaf {
    int32_t   leaf_ID;
    struct t_atm_addr  leaf_address;
}
```

The field *leaf_ID* is a unique identifier for the leaf on the point-to-multipoint connection. When the connection is initially setup via *connect*( ), the remote device is implicitly assigned a leaf identifier of zero. Other leafs added to the connection are assigned a non-zero value by the transport user. The leaf identifier is used as a correlator for the following subsequent operations:

- receiving indications regarding changes in the leaf's status
- removing the leaf from the point-to-multipoint connection.

The values used for *leaf_ID* may be used simultaneously on point-to-multipoint connections other than the connection indicated by parameter *socket*.

Legal values for the field *leaf_ID* are 1 thru (2**15 - 1). This field is mapped to octets 6 thru 6.1 of the ATM Forum's ''Endpoint Reference'' information element, defined in section 5.4.8.1 in referenced document **ATMNAS**.

The field *leaf_address* specifies the network address of the device being added as a leaf of the connection. Legal values for the field *leaf_address* are the same as those defined for option T_ATM_DEST_ADDR in Appendix I (ATM Transport Protocol Information for XTI).

A successful return of this function does not imply that the leaf has been successfully added; rather, it means that the leaf addition request is being processed by the network. Completion (success or failure) of the leaf addition is indicated at a later time. An unsuccessful return of this function with errno set to [EINVAL] is additionally used to indicate the following error conditions:

- the socket type of the specified socket is not a point-to-multipoint connection
- the connection has been aborted
- the connection was forcibly closed by a peer.

### J.3.2 Removing a Leaf

T_ATM_DROP_LEAF

This option is used to initiate an abortive removal of a leaf from an already established point-to-multipoint connection. When this option is invoked, data previously sent via *send*( ) may not be received by the leaf. Note that this option is ‘‘write-only’’; it is a valid parameter of *setsockopt*( ), but not *getsockopt*( ).

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

```
struct t_atm_drop_leaf {
    int32_t  leaf_ID;
    int32_t  reason;
}
```

The field *leaf_ID* identifies the leaf that is being removed from the connection. When the connection is initially setup via *connect*( ), the remote device is implicitly assigned a leaf identifier of zero. Other leafs added to the connection are assigned a non-zero value by the transport user when the leaf is added to the connection.

Legal values for the field *leaf_ID* are 0 thru (2**15 - 1). This field is mapped to octets 6 thru 6.1 of the ATM Forum's "Endpoint Reference" information element, defined in section 5.4.8.1 in the referenced document **ATMNAS**.

The field *reason* specifies the reason for the leaf removal. This field is mapped to octet 6 of the Q.2931 Cause information element.

A successful return of this function does not imply that the leaf has been completely removed; rather, it means that the leaf removal request is being processed by the network. Eventual completion of the leaf removal is assumed. An unsuccessful return of this function with errno set to [EINVAL] is additionally used to indicate the following error conditions:

- the socket type of the specified socket is not a point-to-multipoint connection
- the connection has been aborted
- the connection was forcibly closed by a peer.

### J.3.3 Receiving Indication of a Change in Leaf Status

There must exist an operating system-specific method of notifying the transport user that a status change has occurred for at least one leaf on a given point-to-multipoint connection. When the transport user detects that such a change has occurred for socket, the transport user invokes *getsockopt*( ) with option parameter T_ATM_LEAF_IND . This action should be repeated until field *status* of the returned option value has a value of T_LEAF_NOCHANGE.

T_ATM_LEAF_IND

This option is used to receive indications of status changes for leafs on an already established point-to-multipoint connection. Note that this option is "read-only"; it is a valid parameter of *getsockopt*( ), but not *setsockopt*( ).

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

```
struct t_atm_leaf_ind {
    int32_t  status;
    int32_t  leaf_ID;
    int32_t  reason;
}
```

The field *status* will contain one of these values:

- T_LEAF_NOCHANGE
  no status change has occurred for any leaf associated with this socket.
- T_LEAF_CONNECTED
  a previous request to add a leaf has been successfully completed
- T_LEAF_DISCONNECTED
  signifies that one of these occurred:
  - a previous request to add a leaf was unsuccessful
  - the leaf has initiated a release from the connection
  - the network has dropped the leaf from the connection.

The field *leaf_ID* identifies the leaf whose status has changed. When the connection is initially setup via *connect*( ), the remote device is implicitly assigned a leaf identifier of zero. Other leafs added to the connection are assigned a non-zero value by the transport user when the leaf is added to the connection.

Legal values for the field *leaf_ID* are 0 thru (2**15 - 1). This field is mapped to octets 6 thru 6.1 of the ATM Forum's "Endpoint Reference" information element, defined in section 5.4.8.1 of referenced document **ATMNAS**.

When field *status* has a value of T_LEAF_DISCONNECTED, the field *reason* specifies the reason for the leaf removal. This field is mapped to octet 6 of the Q.2931 Cause information element.

An unsuccessful return of this function with *errno* set to [EINVAL] is additionally used to indicate the following error conditions:

- the socket type of the specified socket is not a point-to-multipoint connection
- the connection has been aborted
- the connection was forcibly closed by a peer.

## J.4 Implementation Notes

This section maps the functions of X/Sockets onto the primitives specified in referenced document **ATMNAS**. This mapping is provided as guidance for the design and development of ATM transport providers.

*accept*( )
The successful return of this function can be mapped to the following ATM Forum primitives:

- *ATM_wait_on_incoming_call* primitive
- *ATM_arrival_of_incoming_call* primitive
- *ATM_accept_incoming_call* primitive
- *ATM_P2P_call_active* primitive (for point-to-point connections)
- *ATM_P2MP_call_active* primitive (for point-to-multipoint connections)

*bind*( )
For transport users passively waiting for incoming connections, this function implements the ATM Forum's *ATM_prepare_incoming_call* primitive. Note that parameter *queue_size* of the ATM Forum's *ATM_prepare_incoming_call* primitive is provided via X/Socket's *listen*( ) function.

*close*( )
The invocation of this function implements the ATM Forum's *ATM_abort_connection* or *ATM_call_release* (request) primitive.

*connect*( )
The invocation of this function implements the ATM Forum's *ATM_connect_outgoing_call* primitive. Additionally, the successful return of this function can be mapped to the following ATM Forum primitives:

- *ATM_P2P_call_active* primitive (point-to-point connection)
- *ATM_P2MP_call_active* primitive (point-to-multipoint connection)

*getpeername*( )
This function implements the ATM Forum's *ATM_query_connection_attributes* primitive.

*getsockname*( )
This function implements the ATM Forum's *ATM_query_connection_attributes* primitive.

*getsockopt*( )
This function implements the ATM Forum's *ATM_query_connection_attributes* primitive. For point-to-multipoint connections, this function also function implements the ATM Forum's *ATM_add_party_success*, *ATM_add_party_reject* and *ATM_drop_party* (Indication) primitives.

*listen*( )
The invocation of this function partially implements the ATM Forum's *ATM_wait_on_incoming_call* primitive, since the queue is enabled. Note that the ATM Forum's primitive can be further classified in this case as a ''polling implementation''.

*recv*( )
The invocation of this function implements the ATM Forum's *ATM_receive_data* primitive.

*recvfrom*( )
The invocation of this function implements the ATM Forum's *ATM_receive_data* primitive.

*recvmsg*( )
The invocation of this function implements the ATM Forum's *ATM_receive_data* primitive.

*send*( )
The invocation of this function implements the ATM Forum's *ATM_send_data* primitive.

*sendmsg*( )
The invocation of this function implements the ATM Forum's *ATM_send_data* primitive.

*sendto*( )
The invocation of this function implements the ATM Forum's *ATM_send_data* primitive.

*setsockopt*( )
This function implements the ATM Forum's *ATM_set_connection_attributes* primitive. For point-to-multipoint connections, this function also function implements the ATM Forum's *ATM_add_party* and *ATM_drop_party* (Request) primitives.

*socket*( )
This function implements the ATM Forum's *ATM_associate_endpoint* primitive.

*Appendix K*

# *ATM Transport Headers*

This Appendix presents the header files **<xti_atm.h>**, **<netatm/atm.h>** and **<_atm_common.h>**, and the proposed additions to **<xti.h>** and **<sys/socket.h>**.

## K.1 Proposed Additions to <xti.h>

```
#define T_LEAFCHANGE     0x0400    /* status of a leaf has changed */

/*
 * The following are new XTI library functions:
 */
/* XTI Library Function: t_addleaf - add a leaf               */
extern int t_addleaf(int32_t, int32_t, struct netbuf *);
/* XTI Library Function: t_removeleaf - remove a leaf         */
extern int t_removeleaf(int32_t, int32_t, int32_t);
/* XTI Library Function: t_rcvleafchange - acknowledge        */
/*                       receipt of a leaf change indication  */
extern int t_rcvleafchange(int32_t, struct t_leaf_status *);
```

## K.2 Proposed Additions to <sys/socket.h>

Add the following macro with distinct integral value in **<sys/socket.h>** in the same name space as AF_UNSPEC:

```
#define AF_ATM           /* ATM transport sockets */
```

## K.3 <xti_atm.h>

```
#include  <_atm_common.h>

/*
 * ATM Levels
 */
#define T_ATM_SIGNALING  0x5301    /* options signalled across UNI */
                                   /* value is not mandatory       */
```

## K.4 <netatm/atm.h>

```
#include  <_atm_common.h>

/*
 * First, the protocol constants.
 */
#define ATM_PROTO_AAL5       0x5301     /* AAL type 5 protocol */
#define ATM_PROTO_SSCOP      0x5302     /* SSCOP protocol      */

/*
 * ATM commonly used constants
 */
#define T_YES                1
#define T_NO                 0
#define T_LEAF_NOCHANGE      0          /* value not mandatory */
#define T_LEAF_CONNECTED     1          /* value not mandatory */
#define T_LEAF_DISCONNECTED  2          /* value not mandatory */


/* ATM-SPECIFIC OPTIONS */

/*
 * ATM signalling-level options
 */
#define T_ATM_SIGNALING      0x5301  /* options signalled across UNI */
                                     /* value is not mandatory       */
#define T_ATM_ADD_LEAF       0x21    /* add leaf to connection       */
                                     /* value is not mandatory       */
#define T_ATM_DROP_LEAF      0x22    /* remove leaf from connection  */
                                     /* value is not mandatory       */
#define T_ATM_LEAF_IND       0x23    /* indication of leaf status    */
                                     /* value is not mandatory       */

/*
 * ATM data structures used for point-to-multipoint connection support
 */
struct t_atm_add_leaf {
    int32_t  leaf_ID;
    struct t_atm_addr  leaf_address;
};

struct t_atm_drop_leaf {
    int32_t  leaf_ID;
    int32_t  reason;
};

struct t_atm_leaf_ind {
    int32_t  status;
    int32_t  leaf_ID;
    int32_t  reason;
};
```

## K.5 <_atm_common.h>

This file should not be included by applications. It is provided so that symbols in it can be exposed through XTI and Socket headers.

```
/*
 * For the purposes of conformance testing, it may be assumed that any
 * constant values defined in these header files are mandatory, unless
 * the constant:
 *   1.  defines an option or options level
 *   2.  is accompanied by a comment that specifies the value is
 *       not mandatory.
 */

/*
 * Leaf status structure.
 */
struct t_leaf_status {
    int32_t  leafid;        /* leaf identifier         */
    int32_t  status;        /* current status          */
    int32_t  reason;        /* reason for leaf removal */
};

/*
 * ATM commonly used constants
 */
#define T_ATM_ABSENT           (-1)
#define T_ATM_PRESENT          (-2)
#define T_ATM_ANY              (-3)
 /*
  * In the 3 constants defined immediately above, the specific value
  * is not mandatory, but any conformin g value must be negative.
  */
#define T_ATM_NULL               0
#define T_ATM_ENDSYS_ADDR        1       /* value is not mandatory */
#define T_ATM_NSAP_ADDR          2       /* value is not mandatory */
#define T_ATM_E164_ADDR          3       /* value is not mandatory */
#define T_ATM_ITU_CODING         0
#define T_ATM_NETWORK_CODING     3


/* ATM-SPECIFIC ADDRESSES */

/*
 * ATM protocol address structure
 */
struct  t_atm_sap {

    struct t_atm_sap_addr {
        int8_t   SVE_tag_addr;
        int8_t   SVE_tag_selector;
        uint8_t  address_format;
        uint8_t  address_length;
        uint8_t  address [20];
    } t_atm_sap_addr;

    struct t_atm_sap_layer2 {
        int8_t   SVE_tag;
        uint8_t  ID_type;
```

```
        union  {
            uint8_t  simple_ID;
            uint8_t  user_defined_ID;
        }  ID;
    } t_atm_sap_layer2;

    struct t_atm_sap_layer3 {
        int8_t   SVE_tag;
        uint8_t  ID_type;
        union  {
            uint8_t  simple_ID;
            int32_t  IPI_ID;
            struct  {
                uint8_t  OUI [3];
                uint8_t  PID [2];
            }  SNAP_ID;
            uint8_t  user_defined_ID;
        }  ID;
    } t_atm_sap_layer3;

    struct t_atm_sap_appl {
        int8_t   SVE_tag;
        uint8_t  ID_type;
        union  {
            uint8_t  ISO_ID [8];
            struct  {
                uint8_t  OUI [3];
                uint8_t  app_ID [4];
            }  vendor_ID;
            uint8_t  user_defined_ID [8];
        }  ID;
    } t_atm_sap_appl;
}


/* ATM-SPECIFIC OPTIONS */

/*
 * ATM signalling-level options
 */
#define T_ATM_AAL5          0x1   /* ATM adaptation layer 5      */
#define T_ATM_TRAFFIC       0x2   /* data traffic descriptor     */
#define T_ATM_BEARER_CAP    0x3   /* ATM service capabilities    */
#define T_ATM_BHLI          0x4   /* higher-layer protocol       */
#define T_ATM_BLLI          0x5   /* lower-layer protocol        */
#define T_ATM_DEST_ADDR     0x6   /* call responder's address    */
#define T_ATM_DEST_SUB      0x7   /* call responder's subaddress */
#define T_ATM_ORIG_ADDR     0x8   /* call initiator's address    */
#define T_ATM_ORIG_SUB      0x9   /* call initiator's subaddress */
#define T_ATM_CALLER_ID     0xa   /* caller's ID attributes      */
#define T_ATM_CAUSE         0xb   /* cause of disconnection      */
#define T_ATM_QOS           0xc   /* desired quality of service  */
#define T_ATM_TRANSIT       0xd   /* choice of public carrier    */

/*
 * T_ATM_AAL5 structure
 */
struct t_atm_aal5 {
    int32_t   forward_max_SDU_size;
```

```
    int32_t   backward_max_SDU_size;
    int32_t   SSCS_type;
};

/*
 * T_ATM_AAL5 values
 */
#define T_ATM_SSCS_SSCOP_REL       1
#define T_ATM_SSCS_SSCOP_UNREL     2
#define T_ATM_SSCS_FR              4

/*
 * T_ATM_TRAFFIC structure
 */
struct t_atm_traffic_substruct {
    int32_t  PCR_high_priority;
    int32_t  PCR_all_traffic;
    int32_t  SCR_high_priority;
    int32_t  SCR_all_traffic;
    int32_t  MBS_high_priority;
    int32_t  MBS_all_traffic;
    int32_t  tagging;
}

struct t_atm_traffic {
    struct   t_atm_traffic_substruct  forward;
    struct   t_atm_traffic_substruct  backward;
    uint8_t  best_effort;
}

/*
 * T_ATM_BEARER_CAP structure
 */
struct t_atm_bearer {
    uint8_t  bearer_class;
    uint8_t  traffic_type;
    uint8_t  timing_requirements;
    uint8_t  clipping_susceptibility;
    uint8_t  connection_configuration;
}

/*
 * T_ATM_BEARER_CAP values
 */
#define T_ATM_CLASS_A        0x01  /* bearer class A                */
#define T_ATM_CLASS_C        0x03  /* bearer class C                */
#define T_ATM_CLASS_X        0x10  /* bearer class X                */
#define T_ATM_CBR            0x01  /* constant bit rate             */
#define T_ATM_VBR            0x02  /* variable bit rate             */
#define T_ATM_END_TO_END     0x01  /* end-to-end timing required    */
#define T_ATM_NO_END_TO_END  0x02  /* end-to-end timing not required */
#define T_ATM_1_TO_1         0x00  /* point-to-point connection     */
#define T_ATM_1_TO_MANY      0x01  /* point-to-multipoint connection */

/*
 * T_ATM_BHLI structure
 */
struct t_atm_bhli {
    int32_t  ID_type;
```

```
        union  {
            uint8_t  ISO_ID [8];
            struct  {
                uint8_t OUI [3];
                uint8_t app_ID [4];
            } vendor_ID;
            uint8_t  user_defined_ID [8];
        } ID;
    }

    /*
     * T_ATM_BHLI values
     */
    #define T_ATM_ISO_APP_ID        0  /* ISO codepoint             */
    #define T_ATM_VENDOR_APP_ID     3  /* vendor-specific codepoint */
    #define T_ATM_USER_APP_ID       1  /* user-specific codepoint   */

    /*
     * T_ATM_BLLI structure
     */
    struct t_atm_blli {
        struct  {
            int8_t  ID_type;
            union  {
                uint8_t  simple_ID;
                uint8_t  user_defined_ID;
            }  ID;
            int8_t  mode;
            int8_t  window_size;
        }  layer_2_protocol;
        struct  {
            int8_t  ID_type;
            union  {
                uint8_t  simple_ID;
                int32_t  IPI_ID;
                struct  {
                    uint8_t  OUI [3];
                    uint8_t  PID [2];
                }  SNAP_ID;
                uint8_t  user_defined_ID;
            }  ID;
            int8_t  mode;
            int8_t  packet_size;
            int8_t  window_size;
        }  layer_3_protocol;
    }

    /*
     * T_ATM_BLLI values
     */
    #define T_ATM_SIMPLE_ID            1   /* ID via ITU encoding    */
    #define T_ATM_IPI_ID               2   /* ID via ISO/IEC TR 9577 */
    #define T_ATM_SNAP_ID              3   /* ID via SNAP            */
    #define T_ATM_USER_ID              4   /* ID via user codepoints */
     /* Constant values in the above 4 definitions are not mandatory */

    #define T_ATM_BLLI_NORMAL_MODE     1
    #define T_ATM_BLLI_EXTENDED_MODE   2
```

```
#define T_ATM_BLLI2_I1745          1   /* I.1745           */
#define T_ATM_BLLI2_Q921           2   /* Q.921            */
#define T_ATM_BLLI2_X25_LINK       6   /* X.25, link layer */
#define T_ATM_BLLI2_X25_MLINK      7   /* X.25, multilink  */
#define T_ATM_BLLI2_LAPB           8   /* Extended LAPB    */
#define T_ATM_BLLI2_HDLC_ARM       9   /* I.4335, ARM      */
#define T_ATM_BLLI2_HDLC_NRM      10   /* I.4335, NRM      */
#define T_ATM_BLLI2_HDLC_ABM      11   /* I.4335, ABM      */
#define T_ATM_BLLI2_I8802         12   /* I.8802           */
#define T_ATM_BLLI2_X75           13   /* X.75             */
#define T_ATM_BLLI2_Q922          14   /* Q.922            */
#define T_ATM_BLLI2_I7776         17   /* I.7776           */

#define T_ATM_BLLI3_X25            6   /* X.25             */
#define T_ATM_BLLI3_I8208          7   /* I.8208           */
#define T_ATM_BLLI3_X223           8   /* X.223            */
#define T_ATM_BLLI3_I8473          9   /* I.8473           */
#define T_ATM_BLLI3_T70           10   /* T.70             */
#define T_ATM_BLLI3_I9577         11   /* I.9577           */

#define T_ATM_PACKET_SIZE_16       4
#define T_ATM_PACKET_SIZE_32       5
#define T_ATM_PACKET_SIZE_64       6
#define T_ATM_PACKET_SIZE_128      7
#define T_ATM_PACKET_SIZE_256      8
#define T_ATM_PACKET_SIZE_512      9
#define T_ATM_PACKET_SIZE_1024    10
#define T_ATM_PACKET_SIZE_2048    11
#define T_ATM_PACKET_SIZE_4096    12

/*
 * ATM network address structure
 */
struct t_atm_addr {
    int8_t   address_format;
    uint8_t  address_length;
    uint8_t  address [20];
}

/*
 * T_ATM_CALLER_ID structure
 */
struct t_atm_caller_id {
    int8_t   presentation;
    uint8_t  screening;
}

/*
 * T_ATM_CALLER_ID values
 */
#define T_ATM_PRES_ALLOWED              0
#define T_ATM_PRES_RESTRICTED           1
#define T_ATM_PRES_UNAVAILABLE          2
#define T_ATM_USER_ID_NOT_SCREENED      0
#define T_ATM_USER_ID_PASSED_SCREEN     1
#define T_ATM_USER_ID_FAILED_SCREEN     2
#define T_ATM_NETWORK_PROVIDED_ID       3

/*
```

```
 * T_ATM_CAUSE structure
 */
struct t_atm_cause {
    int8_t   coding_standard;
    uint8_r  location;
    uint8_r  cause_value;
    uint8_r  diagnostics [4];
}

/*
 * T_ATM_CAUSE values
 */
#define T_ATM_LOC_USER                      0
#define T_ATM_LOC_LOCAL_PRIVATE_NET         1
#define T_ATM_LOC_LOCAL_PUBLIC_NET          2
#define T_ATM_LOC_TRANSIT_NET               3
#define T_ATM_LOC_REMOTE_PUBLIC_NET         4
#define T_ATM_LOC_REMOTE_PRIVATE_NET        5
#define T_ATM_LOC_INTERNATIONAL_NET         7
#define T_ATM_LOC_BEYOND_INTERWORKING      10

#define T_ATM_CAUSE_UNALLOCATED_NUMBER                      1
#define T_ATM_CAUSE_NO_ROUTE_TO_TRANSIT_NETWORK             2
#define T_ATM_CAUSE_NO_ROUTE_TO_DESTINATION                 3
#define T_ATM_CAUSE_NORMAL_CALL_CLEARING                   16
#define T_ATM_CAUSE_USER_BUSY                              17
#define T_ATM_CAUSE_NO_USER_RESPONDING                     18
#define T_ATM_CAUSE_CALL_REJECTED                          21
#define T_ATM_CAUSE_NUMBER_CHANGED                         22
#define T_ATM_CAUSE_ALL_CALLS_WITHOUT_CALLER_ID_REJECTED   23
#define T_ATM_CAUSE_DESTINATION_OUT_OF_ORDER               27
#define T_ATM_CAUSE_INVALID_NUMBER_FORMAT                  28
#define T_ATM_CAUSE_RESPONSE_TO_STATUS_ENQUIRY             30
#define T_ATM_CAUSE_UNSPECIFIED_NORMAL                     31
#define T_ATM_CAUSE_REQUESTED_VPCI_VCI_NOT_AVAILABLE       35
#define T_ATM_CAUSE_VPCI_VCI_ASSIGNMENT_FAILURE            36
#define T_ATM_CAUSE_USER_CELL_RATE_NOT_AVAILABLE           37
#define T_ATM_CAUSE_NETWORK_OUT_OF_ORDER                   38
#define T_ATM_CAUSE_TEMPORARY_FAILURE                      41
#define T_ATM_CAUSE_ACCESS_INFO_DISCARDED                  43
#define T_ATM_CAUSE_NO_VPCI_VCI_AVAILABLE                  45
#define T_ATM_CAUSE_UNSPECIFIED_RESOURCE_UNAVAILABLE       47
#define T_ATM_CAUSE_QUALITY_OF_SERVICE_UNAVAILABLE         49
#define T_ATM_CAUSE_BEARER_CAPABILITY_NOT_AUTHORIZED       57
#define T_ATM_CAUSE_BEARER_CAPABILITY_UNAVAILABLE          58
#define T_ATM_CAUSE_SERVICE_OR_OPTION_UNAVAILABLE          63
#define T_ATM_CAUSE_BEARER_CAPABILITY_NOT_IMPLEMENTED      65
#define T_ATM_CAUSE_INVALID_TRAFFIC_PARAMETERS             73
#define T_ATM_CAUSE_AAL_PARAMETERS_NOT_SUPPORTED           78
#define T_ATM_CAUSE_INVALID_CALL_REFERENCE_VALUE           81
#define T_ATM_CAUSE_IDENTIFIED_CHANNEL_DOES_NOT_EXIST      82
#define T_ATM_CAUSE_INCOMPATIBLE_DESTINATION               88
#define T_ATM_CAUSE_INVALID_ENDPOINT_REFERENCE             89
#define T_ATM_CAUSE_INVALID_TRANSIT_NETWORK_SELECTION      91
#define T_ATM_CAUSE_TOO_MANY_PENDING_ADD_PARTY_REQUESTS    92
#define T_ATM_CAUSE_MANDITORY_INFO_ELEMENT_MISSING         96
#define T_ATM_CAUSE_MESSAGE_TYPE_NOT_IMPLEMENTED           97
#define T_ATM_CAUSE_INFO_ELEMENT_NOT_IMPLEMENTED           99
#define T_ATM_CAUSE_INVALID_INFO_ELEMENT_CONTENTS         100
```

```
#define T_ATM_CAUSE_MESSAGE_INCOMPATIBLE_WITH_CALL_STATE  101
#define T_ATM_CAUSE_RECOVERY ON_TIMER_EXPIRY              102
#define T_ATM_CAUSE_INCORRECT_MESSAGE_LENGTH              104
#define T_ATM_CAUSE_UNSPECIFIED_PROTOCOL_ERROR            111


/*
 * T_ATM_QOS structure
 */
struct t_atm_qos_substruct {
    int32_t  coding_standard;
}

struct t_atm_qos {
    int8_t   coding_standard;
    struct t_atm_qos_substruct  forward;
    struct t_atm_qos_substruct  backward;
}

/*
 * T_ATM_QOS values
 */
#define T_ATM_QOS_CLASS_0     0
#define T_ATM_QOS_CLASS_1     1
#define T_ATM_QOS_CLASS_2     2
#define T_ATM_QOS_CLASS_3     3
#define T_ATM_QOS_CLASS_4     4

/*
 * T_ATM_TRANSIT structure
 */
struct t_atm_transit {
    uint8_t  length;
    uint8_t  network_id[];           /* variable-sized array */
}
```

# Glossary

**abortive release**
An abrupt termination of a transport connection, which may result in the loss of data.

**asynchronous mode**
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.

**CL**
Connectionless (a deprecated synonym for "connectionless-mode").

**CO**
Connection-oriented (a deprecated synonym for "connection-mode").

**connection establishment**
The phase in connection-mode that enables two transport users to create a transport connection between them.

**connection-mode**
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.

**connectionless-mode**
A mode of transfer where different units of data are passed through the network without any relationship between them.

**connection release**
The phase in connection-mode that terminates a previously established transport connection between two users.

**datagram**
A unit of data transferred between two users of the connectionless-mode service.

**data transfer**
The phase in connection-mode or connectionless-mode that supports the transfer of data between two transport users.

**DNS**
The Domain Name System defined in RFC 1035. This system provides translation between host names and Internet addresses.

**EM**
Event Management

**expedited data**
Data that are considered urgent. The specific semantics of expedited data are defined by the transport provider that provides the transport service.

**ETSDU**
Expedited Transport Service Data Unit

**expedited transport service data unit**
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).

**FQDN**
Fully-qualified domain name.

**host byte order**
The implementation-dependent byte order supported by the local host machine (see the Glossary entry for "Network Byte Order"). Functions are provided to convert 16 and 32-bit values between network and host byte order (see *htonl*( )).

**initiator**
An entity that initiates a connection request.

**ISO**
International Organization for Standardization

**legacy**
An item marked ‘‘LEGACY’’ in this specification means that it is being retained for compatibility with older applications, but has limitations which makes it inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality.

**network byte order**
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.

**network host database**
A database whose entries define the names and network addresses of host machines. See *gethostent*( ).

**network net database**
A database whose entries define the names and network numbers of networks. See *getnetent*( ).

**network protocol database**
A database whose entries define the names and protocol numbers of protocols. See *getprotoent*( ).

**network service database**
A database whose entries define the names and local port numbers of services. See *getservent*( ).

**orderly release**
A procedure for gracefully terminating a transport connection with no loss of data.

**OSI**
Open System Interconnection

**responder**
An entity with whom an initiator wishes to establish a transport connection.

**socket**
A communications endpoint associated with a file descriptor that provides communications services using a specified communications protocol.

**SVID**
System V Interface Definition

**synchronous mode**
The mode of execution in which transport service functions wait for specific asynchronous events to occur before returning control to the user.

**TC**
Transport Connection

**TCP**
Transmission Control Protocol

**TLI**
Transport Level Interface

**transport address**
The identifier used to differentiate and locate specific transport endpoints in a network.

**transport connection**
The communication circuit that is established between two transport users in connection-mode.

**transport endpoint**
The communication path, which is identified by a file descriptor, between a transport user and a specific transport provider. A transport endpoint is called passive before, and active after, a relationship is established, with a specific instance of this transport provider, identified by the TSAP.

**transport provider identifier**
A character string used by the function to identify the transport service provider.

**transport service access point**
A TSAP is a uniquely identified instance of the transport provider. A TSAP is used to identify a transport user on a certain endsystem. In connection-mode, a single TSAP may have more than one connection established to one or more remote TSAPs; each individual connection then is identified by a transport endpoint at each end.

**transport service data unit**
A unit of data transferred across the transport service with boundaries and content preserved unchanged. A TSDU may be divided into sub-units passed between the user and XTI. The T_MORE flag is set in all but the last fragment of a TSDU sequence constituting a TSDU. The T_MORE flag implies nothing about how the data is handled and passed to the lower level by the transport provider, and how they are delivered to the remote user.

**transport service provider**
A transport protocol providing the service of the transport layer.

**transport service user**
An abstract representation of the totality of those entities within a single system that make use of the transport service.

**TSAP**
See Transport Service Access Point

**TSDU**
See Transport Service Data Unit

**UDP**
User Datagram Protocol

**user application**
The set of user programs, implemented as one or more process(es) in terms of UNIX semantics, written to realise a task, consisting of a set of user required functions.

**XTI**
X/Open Transport Interface

**XEM**
X/Open Event Management Interface

# Index


<arpa/inet.h> ..... **98**
<fcntl.h> ..... **66**
<netatm/atm.h> ..... 437
<netdb.h> ..... **99**
<net/if.h> ..... **67**
<netinet/in.h> ..... **108**
<netinet/in.h> for IPv6 ..... **115**
<netinet/tcp.h> ..... **103**
<sys/socket.h> ..... **68**, 437
<sys/stat.h> ..... **73**
<sys/uio.h> ..... **74**
<sys/un.h> ..... **106**
<unistd.h> ..... **104**
_XOPEN_SOURCE ..... 3
<xti_atm.h> ..... 437
<xti.h> ..... 161, 188, 242, 317, 437
<xti_inet.h> ..... **261**
<xti_mosi.h> ..... 351
<xti_osi.h> ..... **276**
abortive release ..... 134, 222, 446
accept ..... 272
accept( ) ..... **12**
accept1 ..... 143, 147
accept2 ..... 143, 147
accept3 ..... 143, 147
additions to <sys/socket.h> for ATM ..... 437
additions to <xti.h> for ATM ..... 437
address ..... 121-122, 129-131, 136, 139, 165
..... 169-170, 174-175, 177, 182, 185
..... 188, 192-193, 211-212, 227-228, 265
address information ..... 85
address string ..... 85
addrinfo ..... **101**
addrinfo structure ..... 86, 101
AI_CANONNAME ..... **101**
AI_NUMERICHOST ..... **101**
AI_PASSIVE ..... **101**
application ..... 121-124, 283, 316
applications ..... 161
portability ..... 283, 315
applications portability ..... 121, 316
association-related ..... 270
association-related options ..... 149
asynchronous ..... 148, 190
asynchronous events ..... 124
asynchronous mode ..... 124, 204, 446
ATM addresses ..... 396
ATM levels ..... 437
ATM options ..... 401
ATM transport headers ..... 437
ATM transport protocol for sockets ..... 427
ATM transport protocol for XTI ..... 395
ATM transport provider ..... 395
bind ..... 121, 138, 143, 146, 169, 239
bind( ) ..... **14**
buffer ..... 167, 180, 202
caller ..... 139, 175, 188
Call structure ..... 322
can ..... 1
canonical name ..... 86
character string ..... 121
checksum check ..... 253
child process ..... 122
CL ..... 446
C language
Issue 4 environment ..... 2
close ..... 138, 146-147, 172
close( ) ..... **16**
closed ..... 143, 146-147
cmsghdr ..... **69**
CO ..... 446
Common Usage C ..... 2
compatibility
future ..... 168
compilation environment ..... **3**
connect( ) ..... **17**
connect1 ..... 143, 147
connect2 ..... 143, 147
connection ..... 121, 174, 202, 204, 222, 224
connection establishment ..... 129, 131
..... 174-175, 204, 251, 279, 446
connection indication ..... 145, 169-170, 188, 206
connectionless ..... 274
connectionless-mode ..... 123, 136, 138
..... 146, 211, 213, 227, 270, 282, 446
connection mode ..... 266
connection-mode ..... 123, 129
..... 131, 138, 147, 279-280, 446
connection-mode service ..... 147
connection release ..... 129, 134-135, 251, 279, 446
connection request ..... 164, 188, 204, 222
constants ..... 161

create
transport endpoint ....130
current event ....141, 146, 190
current state ....122, 141, 146, 187, 236
data ....202, 206, 211, 219, 222, 225, 227
datagram ....123, 137, 446
datagram structure ....322
data transfer ....129, 132, 136, 146-147, 279, 446
data unit ....133, 136, 209, 211, 227
discarded ....137
default ....192, 283
de-initialisation ....129-130, 136, 146
descriptive name ....85
device ....283
device driver ....316
discarded data unit ....137
discon ....209, 225
disconnect ....125, 206
indication ....143
disconnection ....129, 148, 206
request ....222
disconnection structure ....321
DNS ....446
dup ....121-122, 236-237
duplex ....132
EAI_AGAIN ....**101**
EAI_BADFLAGS ....**101**
EAI_FAIL ....**101**
EAI_FAMILY ....**101**
EAI_MEMORY ....**102**
EAI_NONAME ....**102**
EAI_SERVICE ....**102**
EAI_SOCKTYPE ....**102**
EAI_SYSTEM ....**102**
EBADF
in recvmsg( ) ....44
EM ....284, 446
endhostent( ) ....**76**
endnetent( ) ....**80**
endprotoent( ) ....**81**
endservent( ) ....**82**
end-to-end significance ....149
enqueue ....131, 139
errmsg ....**177**
errno ....177
errnum ....235
error ....318
error code ....213, 235
error codes ....317-318
TACCES ....317
TADDRBUSY ....317
TBADADDR ....317
TBADDATA ....317
TBADF ....317
TBADFLAG ....317
TBADNAME ....317
TBADOPT ....317
TBADQLEN ....317
TBADSEQ ....317
TBUFOVFLW ....317
TFLOW ....317
TINDOUT ....317
TLOOK ....317
TNOADDR ....317
TNODATA ....317
TNODIS ....317
TNOREL ....317
TNOSTRUCTYPE ....317
TNOTSUPPORT ....317
TNOUDERR ....317
TOUTSTATE ....317
TPROTO ....318
TPROVMISMATCH ....317
TQFULL ....317
TRESADDR ....317
TRESQLEN ....317
TSTATECHNG ....317
TSYSERR ....317
error descriptions ....84
error handling ....123
error indication ....213
error message ....177, 235
error number ....235
error numbers ....6
established
connection ....280
ETSDU ....133, 192-193, 202, 216, 219-221, 446
event ....141, 146, 282
current ....141, 146, 190
event management ....126-127
Event Management ....284
events ....285, 318
accept1 ....143, 147
accept2 ....143, 147
accept3 ....143, 147
bind ....143, 146
closed ....143, 146-147
connect1 ....143, 147
connect2 ....143, 147
incoming ....144
listen ....144, 147, 188
opened ....143, 146

optmgmt ....143, 146
outgoing ....143
pass_conn ....144, 147
rcv ....144, 147, 202
rcvconnect ....144, 147
rcvdis1 ....144, 147
rcvdis2 ....144, 147
rcvdis3 ....144, 147
rcvrel ....144, 147
rcvudata ....144, 146
rcvuderr ....144, 146
snd ....143, 147
snddis1 ....143, 147
snddis2 ....143, 147
sndrel ....143, 147
sndudata ....143, 146
T_CONNECT ....318
T_DATA ....318
T_DISCONNECT ....318
T_EXDATA ....318
T_GODATA ....318
T_GOEXDATA ....318
T_LISTEN ....318
T_ORDREL ....318
T_UDERR ....318
unbind ....143, 146
events and t_look ....125
example ....287
example for select ....295
exec ....236-237
execution mode ....124, 133
expedited data ....129, 133, 193, 202 ....216, 219, 251, 266, 274, 327, 446
expedited transport service data unit ....446
ETSDU ....193, 202
fcntl ....121, 124, 175, 188-189, 202-204 ....211-212, 220, 227-228
fcntl( ) ....**20**
fcntl.h ....192
fd ....121, 143
features ....138-139
F_GETOWN ....20, 66
fgetpos( ) ....**21**
file.c ....161
file descriptor ....121, 172, 182, 192, 236
flag ....192, 195, 202
flags ....192, 195, 202, 274, 318, 321
T_CHECK ....318
T_CURRENT ....318
T_DEFAULT ....318
T_EXPEDITED ....318
T_FAILURE ....318
T_MORE ....318
T_NEGOTIATE ....318
T_NOTSUPPORT ....318
T_PARTSUCCESS ....318
T_PUSH ....318
T_READONLY ....318
T_SUCCESS ....318
flow control ....137
fork ....121-122, 236-237
FQDN ....89, 446
freeaddrinfo( ) ....**85**
freehostent ....**76**
F_SETOWN ....20, 66
fsetpos( ) ....**22**
ftell( ) ....**23**
full duplex ....132
fully-qualified domain name ....89
gai_strerror( ) ....**84**
General purpose defines ....323
getaddrinfo( ) ....**85**
gethostbyaddr( ) ....**76**
gethostbyname( ) ....**76**
gethostent( ) ....**76**
gethostname( ) ....**88**
getipnodebyaddr( ) ....**76**
getipnodebyname( ) ....**76**
getnameinfo( ) ....**89**
getnetbyaddr( ) ....**80**
getnetbyname( ) ....**80**
getnetent( ) ....**80**
getpeername( ) ....**24**
getprotobyname( ) ....**81**
getprotobynumber( ) ....**81**
getprotoent( ) ....**81**
getservbyname( ) ....**82**
getservbyport( ) ....**82**
getservent( ) ....**82**
getsockname( ) ....**25**
getsockopt( ) ....**26**
headers
<xti.h> ....317
h_errno ....**91**
h_errno( ) ....**76**
host byte order ....447
hostent ....**99**
host name ....85
htonl( ) ....**92**
htons( ) ....**92**
if_freenameindex( ) ....**29**
if_indextoname( ) ....**30**

if_nameindex( ).....**31**
if_nametoindex( ).....**32**
implementation-dependent.....1
in_addr.....98, 108
INADDR_ANY .....108
INADDR_BROADCAST.....108
in_addr_t.....98-99
incoming events.....144
inet_addr( ).....**93**
inet_lnaof( ).....**93**
inet_makeaddr( ).....**93**
inet_netof( ).....**93**
inet_network( ).....**93**
inet_ntoa( ).....**93**
inet_pton( ).....**95**
initialisation .....129-130, 136, 146, 192
initiator.....129, 447
in_port_t .....98-99, 108
interfaces
  implementation.....2
  use.....2
internet protocol features .....251
Internet protocol-specific information.....251
IOV_MAX .....318
IP-level Options .....328
IPPORT_RESERVED.....99
IPPROTO_ macros
  defined in <netinet/in.h> .....108
IP_TOS type of service .....329
ISO.....265, 325, 447
  priorities .....325
  protection levels .....326
  transport classes .....325
ISO C .....2
language-dependent .....177
legacy .....1, 447
library functions.....319
library structure .....167
linger.....**70**
listen.....170, 188, 272
listen( ).....**33**
listener application.....122
lseek( ) .....**35**
management options.....268, 271, 327
mandatory features.....283
maximum size
  address.....193
  address buffer .....169, 185
  buffer.....175, 188, 204, 211, 213
  ETSDU.....193, 221
  TSDU.....134, 193, 221, 259
may .....1
memory
  allocate .....167, 180
mode
  asynchronous .....124
  connection.....129, 131, 138, 147, 266, 279-280
  connectionless .....136, 138, 146
  .....211, 213, 227, 270, 282
  record-oriented .....134
  stream-oriented.....134
  synchronous.....124, 190
modes of service .....123
mosi Header File.....351
MSG_ macros
  defined in <sys/socket.h> .....**71**
multiple options.....155
must.....1
name information .....89
name space
  X/Open.....3
native ATM services .....395, 427
netbuf structure.....150, 167, 195
netent.....**99**
network byte order.....447
network host database .....447
network net database.....447
network protocol database.....447
network service database .....447
next state .....146
ntohl( ).....**92**
ntohs( ) .....**92**
NULL .....169
null
  call.....222
null pointer .....168-170, 177, 180, 194, 206, 213
ocnt .....**143**
O_NONBLOCK flag .....124
open.....192
opened .....143, 146
option
  value.....156
option management.....323
option negotiation
  initiate .....152
  response.....153
options
  association-related .....149
  connectionless-mode .....270
  connection mode .....266
  expedited data.....266
  format .....158

generalities....149
illegal....151
ISO-specific....325-326
management....268
multiple....155
privileged....155
quality of service....266
read-only....155
retrieving information....154
TCP-level....252
T_IP-level....254
transport endpoint....195
transport level....119
transport provider....175
T_UDP-level....253
unsupported....152
XTI-level....325
Options management structure....**321**
options with end-to-end significance....149
option values....266
optmgmt....138, 143, 146
orderly release....134, 208, 224, 447
OSI....447
transport classes....273
outgoing events....143
outstanding connection indications...145, 170, 206
pass_conn....144, 147
poll....190
poll( )....**36**
polling....125
portability....160
portable....121, 283, 315
precedence levels
IP....329
primitives....124-125
process....122
program....161
programs
multiple protocol....283
protocol....121, 139, 149, 169, 174
....182, 185, 192, 195, 213, 265, 283
protocol independence....183, 193, 283
protocol-specific servicelimits....320
protoent....**99**
quality of service....266, 270, 327
queue....131, 139, 189
rate....266
rate structure....**326**
rcv....144, 147
rcvconnect....144, 147
rcvdis1....144, 147
rcvdis2....144, 147
rcvdis3....144, 147
rcvrel....144, 147
rcvreldata....144
rcvudata....144
rcvuderr....144
rcvvudata....320
read( )....**37**
readv( )....**37**
reason
disconnection....206
receipt....208
receive....202, 209, 211
Receiving Data....132, 137
record-oriented....134
recv( )....**38**
recvfrom( )....**40**
recvmsg( )....**43**
release....129, 134, 147, 208, 222, 224
reliable....123
remote user....126, 131, 134-135, 172
....175-176, 222, 280, 282
reqvalue....266, 326
reqvalue structure....**326**
resfd....**143**
responder....129, 447
sa_family_t....**68**
safety....6
select( )....**46**
send( )....**47**
Sending Data....133, 137
sendmsg( )....**49**
sendto( )....**52**
servent....**99**
server program....286, 295
service definition
ISO....134, 265, 272
TCP....134
service name....85
service type defines....320
sethostent( )....**76**
setnetent( )....**80**
setprotoent( )....**81**
setservent( )....**82**
setsockopt( )....**55**
shall....1
should....1
shutdown( )....**58**
snd....143, 147-148, 265
snddis1....143, 147
snddis2....143, 147

sndrel....143, 147-148
sndreldata....143
sndudata....143, 146, 148, 265
sockaddr_in ....**108**
sockaddr_un ....**106**
socket ....257-258, 447
socket( ) ....**59**
socketpair( )....**61**
SO_ macros
  defined in <sys/socket.h> ....**70**
standard error....177
state ....141-142, 146, 323
  current....141, 146, 187, 236
  next....146
  T_DATAXFER ....**142**, 323
  T_IDLE ....**142**, 323
  T_INCON ....**142**, 323
  T_INREL ....**142**, 323
  T_OUTCON ....**142**, 323
  T_OUTREL....**142**, 323
  T_UNBIND....**142**
  T_UNBND ....323
  T_UNIT....**142**
state table ....146-147, 282
status
  connection....175
  connection request....132, 204
stream-oriented....134
strerror(3C) ....177
struct netbuf....321
struct rate ....326
struct reqvalue ....326
struct t_bind....321
struct t_call....322
struct t_discon....322
struct thrpt ....326
struct t_info....320
struct t_kpalive ....328
struct t_linger ....325
struct t_opthdr ....321
struct t_optmgmt....321
struct transdel....326
struct t_uderr....322
struct t_unitdata ....322
structure types ....322
  T_BIND....322
  T_CALL....322
  T_DIS ....322
  T_INFO....322
  T_OPTMGMT ....322
  T_UDERROR....322
  T_UNITDATA ....322
SVID ....447
synchronise....236
synchronous mode ....124, 190, 204, 447
T_ABSREQ....**323**
t_accept ....138, 164, 251, 257, 272, 319
t_accept( ) ....**164**
TACCES....**317**
T_ACTIVEPROTECT ....**326**
t_addleaf( ) ....**420**
T_ADDR....**323**
TADDRBUSY....**317**
T_ALL ....**323**
t_alloc....138, 167, 180, 319, 322
t_alloc( ) ....**167**
T_ALLOPT....**323**
t_atm_sap structure ....397
TBADADDR ....**317**
TBADDATA ....**317**
TBADF ....**317**
TBADFLAG ....**317**
TBADNAME....**317**
TBADOPT ....**317**
TBADQLEN ....**317**
TBADSEQ....**317**
t_bind ....121, 138, 169, 239, 257, 272, 319
T_BIND....**322**
t_bind( )....**169**
TBUFOVFLW ....**317**
TC....447
t_call ....164
T_CALL ....**322**
T_CHECK ....**318**
T_CLASS0 ....**325**
T_CLASS1 ....**325**
T_CLASS2 ....**325**
T_CLASS3 ....**325**
T_CLASS4 ....**325**
t_close ....138, 172, 272, 279, 319
t_close( )....**172**
T_CLTS ....**321**
t_connect ....125
T_CONNECT ....125-126
t_connect ....138, 174, 204, 257, 272
T_CONNECT....285, 318
t_connect ....319
t_connect( )....154, 174
T_COTS ....**321**
T_COTS_ORD....**321**
TCP....134, 447
TCP-level options....252, 328

TCP_NODELAY .....103
T_CRITIC_ECP .....**329**
T_CURRENT .....**318**
T_DATA .....125-126, 136, 147-148, 285, 318
T_DATAXFER .....**142**, 147, 323
T_DEFAULT .....**318**
T_DIS .....125, 148, 322
T_DISCONNECT .....125, 127, 133, 148, 285, 318
terminated
  connection .....280
terminology .....**1**
t_errno .....123, 179, 235, 318
t_error .....123, 138, 177, 235, 319
t_error( ) .....**177**
T_EXDATA .....125-126, 258, 285, 318
T_EXPEDITED .....133, 202, 216, 318
T_FAILURE .....**318**
T_FLASH .....**329**
TFLOW .....137, 285, 317
t_free .....138, 180, 319
t_free( ) .....**180**
t_getinfo .....138, 182, 272, 319
t_getinfo( ) .....**182**
t_getprotaddr .....138, 185, 319
t_getprotaddr( ) .....**185**
t_getstate .....138, 187, 319
t_getstate( ) .....**187**
T_GODATA .....125, 127, 136, 285, 318
T_GOEXDATA .....125, 127, 285, 318
T_HIREL .....**329**
T_HITHRPT .....**329**
thread cancellation point .....6
threads .....6
thread safety .....6
thrpt .....266, 326
thrpt structure .....**326**
T_IDLE .....**142**, 146-147, 174, 323
T_IMMEDIATE .....**329**
T_INCON .....**142**, 147, 323
TINDOUT .....**317**
T_INETCONTROL .....**329**
T_INET_IP .....**328**
T_INET_TCP .....328
T_INET_UDP .....328
T_INFINITE .....**323**
T_INFO .....**322**
T_INREL .....**142**, 147, 323
T_INVALID .....**323**
T_IP_BROADCAST .....254, 328
T_IP_DONTROUTE .....254, 328
T_IP-level options .....254
T_IP_OPTIONS .....254, 328
T_IP_REUSEADDR .....255, 328
T_IP_TOS .....255, 328
T_IP_TTL .....256, 328
T_ISO_TP .....**326**
t_kpalive .....**328**
T_LDELAY .....**329**
TLI .....315-316, 448
t_listen .....125
T_LISTEN .....125-126
t_listen .....138, 148
T_LISTEN .....148
t_listen .....**188**, 251, 258, 273
T_LISTEN .....285, 318
t_listen .....319
t_listen( ) .....154, 188
T_LOCOST .....**329**
TLOOK .....125, 133, 148, 317
t_look .....125, 138, 190, 258, 319
t_look( ) .....**190**
T_MORE .....133, 202, 216, 219, 318
T_MORE flag .....251
T_NB_ABORT .....311
T_NB_BCAST_NAME .....307
T_NB_CLOSED .....311
T_NB_GROUP .....307
T_NB_LOCAL .....307
T_NB_NAMELEN .....307
T_NB_NOANSWER .....311
T_NB_OPREJ .....311
T_NB_UNIQUE .....307
T_NEGOTIATE .....**318**
T_NETCONTROL .....**329**
T_NO .....**323**
TNOADDR .....**317**
TNODATA .....**317**
TNODIS .....**317**
T_NOPROTECT .....**326**
TNOREL .....**317**
TNOSTRUCTYPE .....**317**
T_NOTOS .....**329**
TNOTSUPPORT .....**317**
T_NOTSUPPORT .....**318**
TNOUDERR .....**317**
T_NULL .....**323**
t_open .....121, 138, 192, 258, 273, 319
t_open( ) .....**192**
T_OPT .....**323**
t-opthdr .....266
t_optmgmt .....138, 195, 319
T_OPTMGMT .....**322**

t_optmgmt( ).....155, 195
T_ORDREL .....125, 127, 148, 285, 318
T_ORDRELDATA .....**321**
TOS precedence levels.....329
T_OUTCON .....**142**, 147, 323
T_OUTREL .....**142**, 147, 323
TOUTSTATE .....**317**
T_OVERRIDEFLASH .....**329**
T_PARTSUCCESS .....**318**
T_PASSIVEPROTECT .....**326**
T_PRIDFLT .....**326**
T_PRIHIGH .....**325**
T_PRILOW.....**326**
T_PRIMID .....**325**
T_PRIORITY.....**329**
T_PRITOP .....**325**
TPROTO .....**318**
TPROVMISMATCH.....**317**
T_PUSH .....220, 230, 318
TQFULL.....**317**
transdel .....266
transdel structure .....326
transport address .....121, 265, 448
transport classes .....273, 325
transport connection ....121, 131, 172, 182, 224, 448
transport endpoint.....121, 142-143, 156, 169
.....172, 174, 190, 192, 194-195, 236, 239, 448
Transport Level Interface (TLI).....315-316
transport level options .....119
transport provider .....121, 129, 141-142, 146
.....150, 182, 192, 236, 265, 279, 321
transport provider identifier .....**121**, 129, 192, 448
transport service.....119, 265, 279
transport service access point.....448
TSAP .....122
transport service data unit.....448
TSDU.....126, 193, 202, 283
transport service provider.....448
transport service user.....121, 129, 131
.....134, 141-142, 174, 279, 448
transport user actions .....145
t_rcv .....125, 138, 148, 202, 258, 274, 319
t_rcv( ) .....**202**
t_rcvconnect .....138, 148, 204, 258, 274
t_rcvconnect( ) .....154, 204, 319
t_rcvdis.....125, 138, 206, 258, 274, 319
t_rcvdis( ).....**206**
t_rcvleafchange( ) .....**424**
t_rcvrel .....125, 148, 208, 319
t_rcvrel( ) .....**208**
t_rcvreldata.....**209**, 319
t_rcvreldata( ) .....**209**
t_rcvudata .....138, 148, 211, 274, 319
t_rcvudata( ) .....154, 211
t_rcvuderr .....125, 138, 213, 274, 320
t_rcvuderr( ).....155, 213
t_rcvv .....138, 320
t_rcvv( ).....**215**
t_rcvvudata.....138
t_rcvvudata( ) .....**217**
T_READONLY .....**318**
t_removeleaf( ) .....**422**
TRESADDR.....**317**
TRESQLEN .....**317**
T_ROUTINE .....**329**
TSAP .....122, 448
TSDU .....126, 133, 192-193, 202
.....219-221, 227, 251, 265, 274, 448
T_SENDZERO.....**321**
T_SNA_CONNECTION_OUTAGE.....359
T_SNA_CONNECTION_SETUP_FAILURE....359
T_SNA_MAX_LU_LEN .....355
T_SNA_MAX_NETID_LEN.....355
T_SNA_MAX_TPN_LEN .....355
T_SNA_SYSTEM_DISCONNECT.....359
T_SNA_TIMEOUT.....359
T_SNA_USER_DISCONNECT .....359
t_snd.....125, 138, 219, 258-259, 274, 320
t_snd( ) .....**219**
t_snddis .....138, 222, 259, 274, 320
t_snddis( ).....**222**
t_sndrel.....**224**, 320
t_sndrel( ).....**224**
t_sndreldata.....**225**, 320
t_sndreldata( ).....**225**
t_sndudata .....138, 227, 259, 274, 320
t_sndudata( ).....**227**
t_sndv .....138, 320
t_sndv( ).....**229**
t_sndvudata.....138, 320
t_sndvudata( ) .....**232**
TSTATECHNG .....**317**
t_strerror .....138, 235, 320
t_strerror( ) .....**235**
T_SUCCESS .....**318**
t_sync .....122, 138, 236, 320
t_sync( ).....**236**
t_sysconf .....138, 320
t_sysconf( ) .....**238**
TSYSERR .....123, 133, 177, 317
T_TCL_CHECKSUM .....**327**
T_TCL_PRIORITY.....**327**

T_TCL_PROTECTION ..... **327**
T_TCL_RESERRORRATE ..... **327**
T_TCL_TRANSDEL ..... **327**
T_TCO_ACKTIME ..... **327**
T_TCO_ALTCLASS1 ..... **327**
T_TCO_ALTCLASS2 ..... **327**
T_TCO_ALTCLASS3 ..... **327**
T_TCO_ALTCLASS4 ..... **327**
T_TCO_CHECKSUM ..... **327**
T_TCO_CLASS ..... **327**
T_TCO_CONNRESIL ..... **327**
T_TCO_ESTDELAY ..... **327**
T_TCO_ESTFAILPROB ..... **327**
T_TCO_EXPD ..... **327**
T_TCO_EXTFORM ..... **327**
T_TCO_FLOWCTRL ..... **327**
T_TCO_LTPDU ..... **327**
T_TCO_NETEXP ..... **327**
T_TCO_NETRECPTCF ..... **327**
T_TCO_PRIORITY ..... **327**
T_TCO_PROTECTION ..... **327**
T_TCO_REASTIME ..... **327**
T_TCO_RELDELAY ..... **327**
T_TCO_RELFAILPROB ..... **327**
T_TCO_RESERRORRATE ..... **327**
T_TCO_THROUGHPUT ..... **327**
T_TCO_TRANSDEL ..... **327**
T_TCO_TRANSFFAILPROB ..... **327**
T_TCP ..... 251
T_TCP_KEEPALIVE ..... 252, 328
T_TCP_MAXSEG ..... 253, 328
T_TCP_NODELAY ..... 253, 328
T_UDATA ..... **323**
T_UDERR ..... 125, 127, 136, 148, 285, 318
T_UDERROR ..... **322**
T_UDP ..... 251
T_UDP_CHECKSUM ..... 253, 328
T_UDP-level options ..... 253
t_unbind ..... 138
T_UNBIND ..... **142**
t_unbind ..... **239**, 320
t_unbind( ) ..... **239**
T_UNBND ..... 146-147, 323
T_UNIT ..... **142**
T_UNITDATA ..... **322**
T_UNSPEC ..... 158, 323
T_YES ..... **323**
UDP ..... 448
UDP-level options ..... 328
unbind ..... 138, 143, 146, 148
undefined ..... 2
unitdata ..... 211, 227
Unitdata error structure ..... 322
UNIX
  process ..... 122
  versions ..... 315
unspecified ..... 2
user application ..... 129, 136, 448
user data ..... 206, 225
User-Network Interface ..... 427
will ..... 2
write( ) ..... **63**
writev( ) ..... **63**
XEM ..... 448
X/Open name space ..... **3**
XTI ..... 119, 448
  applications ..... 161
  features ..... 138-139
  library ..... 161
XTI_DEBUG ..... **325**
XTI error return ..... 318
XTI_GENERIC ..... **325**
XTI level ..... 325
XTI-level options ..... 199, 325
XTI_LINGER ..... **325**
XTI_RCVBUF ..... **325**
XTI_RCVLOWAT ..... **325**
XTI_SNDBUF ..... **325**
XTI_SNTLOWAT ..... **325**
Zero-length TSDUs and TSDU fragments ..... 183
  ..... 193, 219-220, 265, 274