Technical Standard

Networking Services
Issue 4

[This page intentionally left blank]

X/Open CAE Specification

Networking Services, Issue 4

X/Open Company Ltd.

X/Open CAE Specification

Networking Services, Issue 4

ISBN: 1-85912-049-0
X/Open Document Number: C438

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

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

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

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

List of Figures

List of Tables

Preface

X/Open

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

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

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

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

X/Open Technical Publications

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

There are two types of X/Open specification:

- *CAE Specifications*

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

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

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

- *Preliminary Specifications*

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

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

In addition, X/Open publishes:

- *Guides*

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

- *Technical Studies*

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

- *Snapshots*

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

Versions and Issues of Specifications

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

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

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

Corrigenda

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

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

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

```
request corrigenda; topic index
```

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

This Document

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

This document centres around three sets of networking interfaces:

1. X/Open Transport Interface (XTI)

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

2. Sockets

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

3. IP Address Resolution

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

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

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

Structure

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

Revision History

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

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

Zero-length TSDUs and TSDU fragments

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

T_MORE

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

Protocol options

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

Options and management structures

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

Expedited Data

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

Connect semantics

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

Add function *t_getprotaddr*( )

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

Add function *t_strerror*( )

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

Add Valid States to function descriptions

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

Add new error codes

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

A number of minor changes were also made, including:

— clarification of the use of the term *socket* in the TCP case

— clarification of support for automatic generation of addresses

— clarification of the management of flow control

— clarification of the significant differences between transport providers

— clarification of the issue of non-guaranteed delivery of data at connection close

— clarification of the ways in which error indications may be received in connectionless working

— enhancement of *t_optmgmt*( ) to allow retrieval of current value of transport provider options

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

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

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

   Section 2.9.1
   The *Protocol options* and *Options and management structures* paragraphs were deleted and replaced with the following:

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

   Section 4.5
   The row for *optmgmt* was deleted from Figure 5, and a new row added to Figure 7 for the event *optmgmt*, as follows:

optmgmt	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL	T_UNBND

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

   Chapter 6, *t_accept*( )
   In the second paragraph, the phrase *protocol-specific parameters* was replaced with *options*.

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

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

   Chapter 6, *t_connect*( )
   In the sixth paragraph, ‘‘If used, *sndcall->opt.buf* must point to the corresponding options structures (**isoco_options** or **tcp_options**);’’ was replaced with ‘‘If used, *sndcall->opt.buf* must point to a buffer with the corresponding options;’’.

   Chapter 6, *t_listen*( )
   In the second paragraph, *protocol-specific parameters* was replaced with *options*.

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

   Chapter 6, *t_rcvconnect*( )
   In the third paragraph, *protocol-specific information* was replaced with *options*.

   Chapter 6, *t_rcvudata*( ) and *t_rcvuderr*( )
   In the third paragraph, *protocol-specific options* was replaced with *options*.

   Chapter 6, *t_sndudata*( )
   In the second paragraph, *protocol-specific options* was replaced with *options*.

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

Appendix B

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

Appendix F

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

Typographical Conventions

The following typographical conventions are used throughout this document:

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

Trade Marks

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

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

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

SNA is a product of International Business Machines Corporation.

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

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

X/Open® is a registered trade mark, and the ''X'' device is a trade mark, of X/Open Company Limited.

Acknowledgements

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

Referenced Documents

The following documents are referenced in this specification:

ACSE

ISO 8649

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

Technical Corrigendum 1: 1990 to ISO 8649: 1988
Amendment 1: 1990 to ISO 8649: 1988
Authentication during association establishment.
Amendment 2: 1991 to ISO 8649: 1988
Connectionless-mode ACSE Service.

ISO 8650

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

Technical Corrigendum 1: 1990 to ISO 8650: 1988
Amendment 1: 1990 to ISO 8650: 1988
Authentication during association establishment.

ISO / IEC 10035

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

Presentation

ISO 8822

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

ISO 8823

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

ISO 8824

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

BER

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

ISO / IEC 9576

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

Session

ISO 8326

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

ISO 8327

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

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

Other References

Minimal OSI

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

ISO 7498

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

ISO Transport

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

TCP

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

UDP

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

TLI Specifications

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

NetBIOS

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

SNA

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

CURL

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

XSH, Issue 4, Version 2

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

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

Chapter 1

Common Information

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

1.1 Terminology

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

The following terms are used in this specification:

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

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

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

must
This describes a requirement on the application or user.

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

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

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

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

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

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

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

1.1.1 Shaded Text

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

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

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

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

1.2 Use and Implementation of Interfaces

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

Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow. 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. Any function declared in a header may also be implemented as a macro defined in the header, so a library function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro. The use of the C-language **#undef** construct to remove any such macro definition will also ensure that an actual function is referred to. Any invocation of a library function that is implemented as a macro will expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called.

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

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 before inclusion of any header. This is needed to enable the functionality described in this document (but see also Section 1.3.1), 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 1.2 on page 2 or Section 1.3.2 on page 5. These **#undef** directives must follow all **#include** directives of any XSI headers.

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

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

The *c89* and *cc* utilities 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 X/Open UNIX Extension

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

1.3.2 The X/Open Name Space

All identifiers in this document are defined in at least one of the headers, as shown in Chapter 9, Chapter 11 and Appendix F. 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_	
<errno.h>	E	
<netdb.h>	h_, n_, p_, s_	
<netinet/in.h>	in_, s_, sin_	
<sys/socket.h>	sa_, if_, ifc_, ifru_, infu_, ifra_, msg_, cmsg_, l_	
<sys/un.h>	sun_	
<xti.h>	l_, t_, T	
ANY header		_t

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

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.

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

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

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 Relationship to Emerging Formal Standards

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

Chapter 2

General Introduction to the XTI

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 specification describe interfaces, parameters and semantics constant across all transport providers. The remainder of the document consists of appendices that provide valuable 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-oriented or connectionless)
  - — 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-oriented) 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 C 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 (Appendix C), Minimum OSI Functionality (Appendix H), SNA Transport provider (Appendix I) and comparison of XTI to TLI (Appendix E) belong to this category.

Some other appendices, however, have evolved into a prescriptive specification, as in the case of Appendix A for the ISO transport provider, Appendix B for the Internet transport provider and Appendix D for the NetBIOS transport provider. 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 main XTI specification, thus becoming brandable XTI options. Support for these transport providers is declared by vendors through the XTI Conformance Statement Questionnaire.

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

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 (EM) interface. Such EM 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 Appendix C provides some guidelines for using EM in XTI applications, a complete specification defining an EM interface is beyond the scope of this document.

Chapter 3

Explanatory Notes for XTI

3.1 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.

3.2 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 connect indications from several transport providers must open a transport endpoint for each provider and listen for connect indications on each of the associated file descriptors.

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

3.4 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.

3.5 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.

3.6 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. An external integer, *t_errno*, which is defined in the header **<xti.h>**, holds the specific error number when such a failure occurs. This value is set when errors occur but is not cleared on successful library calls, so it should be tested only after an error has been indicated. If implemented, 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 unusable. To continue in this case, all users of the *fd* must close it. Then the transport endpoint may be re-opened and initialised.

3.7 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 disconnect indication arrives. The user must somehow be informed that the connection has been broken.

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

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 3.8 on page 16.

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 the 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 clears it. 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 3-1 summarises this.

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

Table 3-1 Events and *t_look*( )

---

* 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.

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

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

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

T_EXDATA
: Expedited data was received by the transport provider.

T_DISCONNECT
: In connection mode only; a disconnect request was received by the transport provider. It may be reported on both data transfer functions and 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*( ) 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 4

XTI Overview

4.1 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 C.1 on page 207, and the figure in Section C.2 on page 208, 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 Class 0
   - negotiation as to its use has been agreed between the two peer transport providers.

   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 disconnect 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.

4.1.1 Initialisation/De-initialisation Phase

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

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

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

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

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

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

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

t_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(s) associated with the given transport endpoint.

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

4.1.2 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 connect indications (only if the provider is able to accept more than one outstanding connect indication; this mode of operation is declared at the time of calling *t_bind*( ) by setting *qlen* greater than 0). However, if more than one endpoint is bound to the same address, only one of them may be used in this way.
- The *t_listen*( ) function is used to look for an enqueued connect indication; if it finds one (at the head of the queue), it returns details of the connect indication, and a local sequence number which uniquely identifies this indication, or it may return a negative value with *t_errno* set to [TNODATA]. The number of outstanding connect requests to dequeue is limited by the value of the *qlen* parameter accepted by the transport provider on the *t_bind*( ) call.
- If the endpoint has more than one connect indication enqueued, the user should dequeue all connect indications (and disconnect indications) before accepting or rejecting any or all of them. The number of outstanding connect indications is limited by the value of the *qlen* parameter accepted by the transport provider on the call to *t_bind*( ).
- When accepting a connect indication, the transport service user may issue the accept on the same (listening) endpoint or on a different endpoint.

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

  If a different endpoint is used, the listening endpoint can continue to receive and enqueue incoming connect requests.
- If the user issues a *t_connect*( ) on a listening endpoint, again, that endpoint can no longer be used to receive and enqueue incoming connect requests.
- A connect 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 connect request. If the request was accepted, the connection establishment phase will be complete on return from this function. This function is used in conjunction with *t_connect*( ) to establish a connection in an asynchronous manner.

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

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

4.1.3 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. Two functions have been defined to support data transfer in connection mode as follows:

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

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

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

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

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 63.

4.1.4 Overview of Connection Release

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

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

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

The orderly release capability is an optional feature of TCP. If supported by the TCP transport provider, orderly release may be 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.

The functions that support connection release are:

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

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

t_sndrel( ) 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.

4.2 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 C.1 on page 207, and the figure in Section C.3 on page 210, that defines the legal sequence in which functions from each phase may be issued.

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

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

4.2.1 Initialisation/De-initialisation Phase

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

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

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

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

Receiving Data

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

- Asynchronous Mode

  The call returns immediately indicating failure. The user must either retry the call repeatedly, or ‘‘poll’’ for incoming data by using the 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 returns when sending a new datagram becomes possible again. A process which sends data in synchronous mode may be blocked for some time.

- Asynchronous Mode

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

4.3 XTI Features

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

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

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

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

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

t_rcvudata( )
t_rcvuderr( )
t_sndudata( )

Mandatory mechanisms:

- synchronous mode
- asynchronous mode.

Utility functions:

t_alloc( )
t_free( )
t_error( )
t_getprotaddr( )
t_getinfo( )
t_getstate( )
t_optmgmt( )
t_strerror( )

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

Optional mechanisms:

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

4.3.1 XTI Functions versus Protocols

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

Functions	**Necessary for Protocol**		**Utility Functions**	
	Connection Oriented	**Connectionless**	**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_rcvconnect( )	x			
t_rcvdis( )	x			
t_rcvrel( )	x			
t_rcvudata( )		x		
t_rcvuderr( )		x		
t_snd( )	x			
t_snddis( )	x			
t_sndrel( )	x			
t_sndudata( )		x		
t_strerror( )			x	
t_sync( )			x	
t_unbind( )	x	x		

Table 4-1 Classification of the XTI Functions

Chapter 5

States and Events in XTI

Table 5-1 through Table 5-7 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 5-5, Table 5-6 and Table 5-7. 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*( ) and *t_strerror*( ) functions have been excluded from the state table because they do not affect the state of the interface.

5.1 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 5-1 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 and final state of interface	T_COTS T_CLTS T_COTS_ORD
T_UNBND	unbound	T_COTS T_COTS_ORD T_CLTS
T_IDLE	no connection established	T_COTS T_COTS_ORD T_CLTS
T_OUTCON	outgoing connection pending for active user	T_COTS T_COTS_ORD
T_INCON	incoming connection pending for passive user	T_COTS T_COTS_ORD
T_DATAXFER	data transfer	T_COTS T_COTS_ORD
T_OUTREL	outgoing orderly release (waiting for orderly release indication)	T_COTS_ORD
T_INREL	incoming orderly release (waiting to send orderly release request)	T_COTS_ORD

Table 5-1 Transport Interface States

5.2 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 5-2, 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 connect indications (connect indications passed to the user but not accepted or rejected).

fd File descriptor of the current transport endpoint.

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

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

Table 5-2 Transport Interface Outgoing Events

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

5.3 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 5-3, 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 connect indications on the current transport endpoint.

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

Table 5-3 Transport Interface Incoming Events

5.4 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 5-4.

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

Table 5-4 Transport Interface User Actions

5.5 State Tables

Table 5-5, Table 5-6 and Table 5-7 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 5-4 on page 31). 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 5-5 Initialisation / De-initialisation States

state event	**T_IDLE**
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

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

state event	**T_IDLE**	**T_OUTCON**	**T_INCON**	**T_DATAXFER**	**T_OUTREL**	**T_INREL**	**T_UNBND**
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_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL	T_UNBIND
closed	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	

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

5.6 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[1]
t_listen( )	T_DISCONNECT[2]
t_rcv( )	T_DISCONNECT, T_ORDREL[3]
t_rcvconnect( )	T_DISCONNECT
t_rcvrel( )	T_DISCONNECT
t_rcvudata( )	T_UDERR
t_snd( )	T_DISCONNECT, T_ORDREL
t_sndudata( )	T_UDERR
t_unbind( )	T_LISTEN, T_DATA[4]
t_sndrel( )	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 3-1.

---

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

Chapter 6

The Use of Options in XTI

6.1 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.

All options have default values. Their values have meaning to and are defined by the protocol level in which they apply. However, their 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 the proposed one, that is, a delay may become longer, or a throughput may become lower.

It is useful to differentiate between options that are *association-related*[5] 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-oriented 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 that are not association-related 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 IP *time-to-live* field, or TCP_NODELAY (see Appendix B on page 199). 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 association-related options 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.

The transport provider has a default value for each option it supports. 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.

---

5. The term ‘‘association’’ is used to denote a pair of communicating transport users.

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 189 and Appendix B on page 199). General XTI options are described in *t_optmgmt*( ) on page 76.

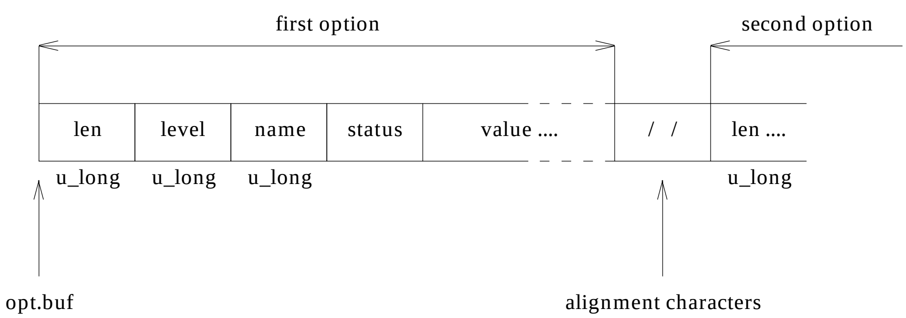
6.2 The Format of Options

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 as 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 6.3.5 on page 40 and *t_optmgmt*( ) on page 76).

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

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



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

6.3 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.

6.3.1 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.

6.3.2 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 76, Appendix A on page 189 and Appendix B on page 199.)

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

- A call to *t_optmgmt*( ) fails with [TBADOPT].
- *t_accept*( ) or *t_connect*( ) fail either with [TBADOPT], or the connection establishment aborts, depending on the implementation and the time the illegal option is detected. If the connection aborts, a T_DISCONNECT event occurs, and 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] in that case.
- A call to *t_sndudata*( ) either fails with [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 76).

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 *level* field of the option.

6.3.3 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 76, Appendix A on page 189 and Appendix B on page 199). 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 6.4 on page 42).

  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 6.4 on page 42). 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* association-related options 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 6.4 on page 42).

6.3.4 Responding to a Negotiation Proposal

In connection-oriented communication, some protocols give the peer transport users the opportunity to negotiate characteristics of the transport connection to be established. These characteristics are association-related options. With the connect 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 6.4 on page 42).

The response options can either be specified with the *t_accept*( ) call, or can be preset for the responding endpoint (not the listening endpoint!) *resfd* in a *t_optmgmt*( ) call (action T_NEGOTIATE) prior to *t_accept*( ) (see Section 6.4 on page 42). 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 will 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 6.3.3 on page 38). Unsupported options are ignored.

6.3.5 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, 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 association-related options that were received with the connection response and the negotiated values of those non-association-related options 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, IP options) that is not subject to negotiation is always set to T_SUCCESS.

t_listen( )

The received association-related options 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, will usually not return the same values.

The number of received options may be variable for subsequent connect indications, since many association-related options are only transmitted on explicit demand by the calling user (for example, 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 association-related options 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 connect indication or an incoming datagram. A detailed description is given in *t_optmgmt*( ) on page 76.

6.3.6 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]. 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.) It depends on timing and implementation conditions whether a *t_accept*( ) call still succeeds or fails with [TLOOK].
- 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 6.4 on page 42).

6.4 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 6.3 on page 37.

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

As usual, association-related options 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 6.3.3 on page 38 and Section 6.3.4 on page 39, 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.

6.5 Supplements

This section contains supplementary remarks and a short summary.

6.5.1 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 connect 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 TCO_THROUGHPUT in Appendix A on page 189). 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.

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

6.5.2 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* possibly diminish as soon as the T_DATAXFER state is entered. Calling *t_optmgmt*( ) does not influence these values (see *t_optmgmt*( ) on page 76).

6.5.3 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 long-word boundary.
- There are options that are association-related and options that are not. On output, the functions *t_listen*( ) and *t_rcvudata*( ) return association-related options 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.

6.6 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 7

XTI Library Functions and Parameters

7.1 How to Prepare XTI Applications

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

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

The syntax for shared libraries is implementation-dependent.

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

7.2 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.

7.3 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 5.6 on page 34.

NAME

t_accept - accept a connect request

SYNOPSIS

```
#include <xti.h>

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

CAVEATS

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

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

VALID STATES

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

ERRORS

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

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

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

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

RETURN 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>

char *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 this way the appropriate size information can be accessed. If the size value associated with any specified field is −1 or −2 (see *t_open*( ) or *t_getinfo*( )), *t_alloc*( ) will be unable to determine the size of the buffer to allocate and will fail, setting *t_errno* to [TSYSERR] and *errno* to [EINVAL]. For any field not specified in *fields*, *buf* will be set to the null pointer and *len* and *maxlen* will be set to zero.

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

VALID STATES

ALL - apart from T_UNINIT

ERRORS

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

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

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

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

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

RETURN 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, struct t_bind *req, struct t_bind *ret);
```

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

VALID STATES

T_UNBND

ERRORS

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

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

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

RETURN 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_close*( ), *t_open*( ), *t_optmgmt*( ), *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.

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

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; the *close*( ) will be abortive if there are no other descriptors in this, or in another process which references the transport endpoint, and in this case will break any transport connection that may be associated with that endpoint.

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

VALID STATES

ALL - apart from T_UNINIT

ERRORS

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

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

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

RETURN 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, struct t_call *sndcall, struct t_call *rcvcall);
```

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

By default, *t_connect*( ) executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (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 connect request to the destination transport user. The *t_rcvconnect*( ) function is used in conjunction with *t_connect*( ) to determine the status of the requested connection.

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

VALID STATES

T_IDLE

ERRORS

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

[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TNODATA]	O_NONBLOCK was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.
[TBADADDR]	The specified protocol address was in an incorrect format or contained illegal information.
[TBADOPT]	The specified protocol options were in an incorrect format or contained illegal information.
[TBADDATA]	The amount of user data specified was not within the bounds allowed by the transport provider.
[TACCES]	The user does not have permission to use the specified address or options.

[TBUFOVFLW] The number of bytes 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.

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

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

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

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

RETURN 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(char *errmsg);
```

DESCRIPTION

Parameters	Before call	After call
errmsg	x	/

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

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

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

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

EXAMPLE

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

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

The diagnostic message to be printed would look like:

```
t_connect failed on fd2: incorrect 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_free - free a library structure

SYNOPSIS

```
#include <xti.h>

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

DESCRIPTION

Parameters	Before call	After call
ptr	x	/
struct_type	x	/

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

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

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

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

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

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

VALID STATES

ALL - apart from T_UNINIT

ERRORS

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

[TSYSERR]	A system error has occurred during execution of this function.
[TNOSTRUCTYPE]	Unsupported *struct_type* requested.
[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI *(t_errno).*

RETURN 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:

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

The values of the fields have the following meanings:

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

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

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

etsdu — A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider. Note that the semantics of expedited data may be quite different for different transport providers (see Appendix A on page 189 and Appendix B on page 199).

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

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

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

flags — This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates that the underlying transport provider supports the sending of zero-length TSDUs. See Appendix A on page 189 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 affect on the values returned by *t_getinfo*( )). These values will only change from the values presented to *t_open*( ) after the endpoint enters the T_DATAXFER state.

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

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

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

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

VALID STATES

ALL - apart from T_UNINIT

ERRORS

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

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

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

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

RETURN 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->maxlen	x	/
boundaddr->addr.len	/	x
boundaddr->addr.buf	?	(?)
boundaddr->qlen	/	/
peeraddr->maxlen	x	/
peeraddr->addr.len	/	x
peeraddr->addr.buf	?	(?)
peeraddr->qlen	/	/

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

VALID STATES

ALL - apart from T_UNINIT

ERRORS

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

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

RETURN 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.
[TSTATECHNG]	The transport provider is undergoing a transient state change.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI *(t_errno)*.

RETURN VALUE

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

T_UNBND	Unbound.
T_IDLE	Idle.
T_OUTCON	Outgoing connection pending.
T_INCON	Incoming connection pending.
T_DATAXFER	Data transfer.
T_OUTREL	Outgoing orderly release (waiting for an orderly release indication).
T_INREL	Incoming orderly release (waiting to send an orderly release request).

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

SEE ALSO

t_open( ).

CHANGE HISTORY

Issue 4

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

NAME

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

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

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

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

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

VALID STATES

T_IDLE, T_INCON

ERRORS

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

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

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

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

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

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

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

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

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

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

CAVEATS

Some transport providers do not differentiate between a connect indication and the connection itself. If this is the case, a successful return of *t_listen*( ) indicates an existing connection (see Appendix B on page 199).

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 5.6 on page 34.

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

VALID STATES

ALL - apart from T_UNINIT

ERRORS

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

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

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

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

RETURN VALUE

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

T_LISTEN Connection indication received.

T_CONNECT Connect confirmation received.

T_DATA Normal data received.

T_EXDATA Expedited data received.

T_DISCONNECT Disconnect received.

T_UDERR Datagram error indication.

T_ORDREL Orderly release indication.

T_GODATA Flow control restrictions on normal data flow 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(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:

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

The values of the fields have the following meanings:

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

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

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

etsdu — A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of –1 specifies that there is no limit on the size of an ETSDU; and a value of –2 specifies that the transfer of expedited data is not supported by the transport provider. Note that the semantics of expedited data may be quite different for different transport providers (see Appendix A on page 189 and Appendix B on page 199).

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

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

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

flags — This is a bit field used to specify other information about the transport provider. If the T_SENDZERO bit is set in flags, this indicates the underlying transport provider supports the sending of zero-length TSDUs. See Appendix A on page 189 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 for *etsdu*, *connect* and *discon*.

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

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

VALID STATES

T_UNINIT

ERRORS

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

[TBADFLAG] — An invalid flag is specified.

[TBADNAME] — Invalid transport provider name.

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

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

RETURN VALUES

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

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, struct t_optmgmt *req, struct t_optmgmt *ret);
```

DESCRIPTION

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

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

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

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

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

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

Each option in the options buffer is 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.

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

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

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

Chapter 6 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 are not association-related (see **Chapter 5**, **The Use of Options**). They may be negotiated in all XTI states except T_UNINIT.

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

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

Table 7-1 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 {
    long l_onoff;    /* switch option on/off     */
    long l_linger;   /* linger period in seconds */
}
```

Legal values for the field *l_onoff* are:

T_NO switch option off
T_YES activate option

The value *l_onoff* is an absolute requirement.

The field *l_linger* determines the linger period in seconds. The transport user can request the default value by setting the field to T_UNSPEC. The default timeout value depends on the underlying transport provider (it is often T_INFINITE). 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.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TACCES]	The user does not have permission to negotiate the specified options.
[TBADOPT]	The specified options were in an incorrect format or contained illegal information.
[TBADFLAG]	An invalid flag was specified.
[TBUFOVFLW]	The number of bytes allowed for an incoming argument *(maxlen)* is greater than 0 but not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI *(t_errno)*.
[TNOTSUPPORT]	This action is not supported by the transport provider.

RETURN VALUE

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

SEE ALSO

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

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

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

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

VALID STATES

T_DATAXFER, T_OUTREL

ERRORS

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

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

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

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

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

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

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

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

RETURN 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 connect 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 connect request and is used in conjunction with *t_connect*( ) to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

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

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

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

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

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

VALID STATES

T_OUTCON

ERRORS

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

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

[TBUFOVFLW] The number of bytes allocated for an incoming argument *(maxlen)* is greater than 0 but not sufficient to store the value of that argument, and the connect information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to T_DATAXFER.

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

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

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

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

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

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

RETURN 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 disconnect

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 disconnect and to retrieve any user data sent with the disconnect. The argument *fd* identifies the local transport endpoint where the connection existed, and *discon* points to a **t_discon** structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

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

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

VALID STATES

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

ERRORS

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

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

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

[TBUFOVFLW] The number of bytes allocated for incoming data *(maxlen)* is greater than 0 but not sufficient to store the data. If *fd* is a passive endpoint with *ocnt* > 1, it remains in state T_INCON; otherwise, the endpoint state is set to T_IDLE.

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

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

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

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

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

VALID STATES

T_DATAXFER,T_OUTREL

ERRORS

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

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

[TNOREL] No orderly release indication currently exists on the specified transport endpoint.

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

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

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

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

[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 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_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, and *flags* is set on return to indicate that the complete data unit was not received. The argument *unitdata* points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

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

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

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

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

VALID STATES

T_IDLE

ERRORS

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

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

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

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

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

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

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

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

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

RETURN 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;
long error;
```

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

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

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

VALID STATES

T_IDLE

ERRORS

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

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

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

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

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

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

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

RETURN 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_snd - send data or expedited data over a connection

SYNOPSIS

```
#include <xti.h>

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

DESCRIPTION

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

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

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

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

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

The sending of a zero-length fragment of a TSDU or ETSDU is only permitted where this is used to indicate the end of a TSDU or ETSDU; 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 189 for a fuller explanation.

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

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

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

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

VALID STATES

T_DATAXFER, T_INREL

ERRORS

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

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

[TBADDATA] Illegal amount of data:

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

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

— 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).

[TBADFLAG] An invalid flag was specified.

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

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

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

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

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

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

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

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 disconnect request

SYNOPSIS

```
#include <xti.h>

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

DESCRIPTION

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

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

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

The values in *call* have different semantics, depending on the context of the call to *t_snddis*( ). When rejecting a connect request, *call* must be non-null and contain a valid value of *sequence* to uniquely identify the rejected connect indication to the transport provider. The *sequence* field is only meaningful if the transport connection is in the T_INCON state. The *addr* and *opt* fields of *call* are ignored. In all other cases, *call* need only be used when data is being sent with the disconnect request. The *addr*, *opt* and *sequence* fields of the **t_call** structure are ignored. If the user does not wish to send data to the remote user, the value of *call* may be a null pointer.

The *udata* structure specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider, as returned in the *discon* field, of the *info* argument of *t_open*( ) or *t_getinfo*( ). If the *len* field of *udata* is zero, no data will be sent to the remote user.

VALID STATES

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

ERRORS

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

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

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

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

[TBADSEQ] An invalid sequence number was specified, or a null *call* pointer was specified, when rejecting a connect request.

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

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

[TLOOK] An asynchronous event, 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 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 disconnect. Therefore a *t_snddis*( ) issued on a connection endpoint may cause data previously sent via *t_snd*( ), or data not yet received, to be lost (even if an error is returned).

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	/

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

VALID STATES

T_DATAXFER,T_INREL

ERRORS

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

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

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

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

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

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

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

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

RETURN 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_sndudata - send a data unit

SYNOPSIS

```
#include <xti.h>

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

DESCRIPTION

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

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

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

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

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

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

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

If it is not possible for the transport provider to immediately detect the conditions that cause the errors [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.

VALID STATES

T_IDLE

ERRORS

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

[TBADDATA] Illegal amount of data. A single send was attempted specifying a TSDU greater than that specified in the *info* argument, or a send of a zero byte TSDU is not supported by the provider.

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

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

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

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

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

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

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

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

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

RETURN 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_strerror - produce an error message string

SYNOPSIS

```
#include <xti.h>

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 implementation-defined. If it is English, the error message string describing the value in *t_errno* is identical to the comments following the *t_errno* codes defined in **<xti.h>**. If an error code is unknown, and the language is English, *t_strerror*( ) returns the string:

```
"<error>: error 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.
[TSTATECHNG]	The transport endpoint is undergoing a state change.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI *(t_errno)*.

RETURN 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_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.
[TOUTSTATE]	The function was issued in the wrong sequence.
[TLOOK]	An asynchronous event has occurred on this transport endpoint.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI *(t_errno)*.

RETURN 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 8

Sockets Interfaces

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.

The associated headers are documented in Chapter 9.

8.1 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.

This specification covers local UNIX connections and Internet protocols.

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.

NAME

accept — accept a new connection on a socket

SYNOPSIS

UX

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

int accept (int socket, struct sockaddr *address, size_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 **size_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 [EWOULDBLOCK] or [EAGAIN].

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 nonnegative 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:

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

[ECONNABORTED] — A connection has been aborted.

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

[EOPNOTSUPP] — The socket type of the specified socket does not support accepting connections.

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

[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.

The *accept*( ) function may fail if:

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

[ENOBUFS] No buffer space is available.

[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>**.

CHANGE HISTORY

First released in Issue 4.

NAME

bind — bind a name to a socket

SYNOPSIS

UX

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

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

DESCRIPTION

The *bind*( ) function assigns an *address* to an unnamed socket. Sockets created with *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.

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:

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

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

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

[EADDRINUSE] The specified address is already in use.

[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.

[EACCES] The specified address is protected and the current user does not have permission to bind to it.

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

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

[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.

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

[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.

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

[EIO] An I/O error occurred.

[EROFS] The name would reside on a read-only filesystem.

The *bind*( ) function may fail if:

[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>**.

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

UX If *fildes* refers to a socket, *close*( ) causes the socket to be destroyed. If the socket is connection-oriented, and the SOCK_LINGER option is set for the socket, 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

UX

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

int connect(int socket, const struct sockaddr *address,
     size_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 initiating socket is not connection-oriented, 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-oriented, 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.

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.

[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-oriented 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:

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

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

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

[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*.

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

The *connect*( ) function may fail if:

[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

UX 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

UX 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

UX 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

UX 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

UX 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

UX

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

int getpeername(int socket, struct sockaddr *address,
                size_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.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[ENOTCONN]	The socket is not connected or otherwise has not had the peer prespecified.
[EINVAL]	The socket has been shut down.
[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

UX

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

int getsockname(int socket, struct sockaddr *address,
                size_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.

[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

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

int getsockopt(int socket, int level, int option_name, void *option_value,
     size_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 protocol number 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 the protocol number of TCP, as defined in the <**netinet/in.h**> header, or as determined by using *getprotobyname*( ) 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.

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

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

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.

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.

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.

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.

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, *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.

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

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

[EINVAL] — The specified option is invalid at the specified socket level.

[EOPNOTSUPP] — The operation is not supported by the socket protocol.

The *getsockopt*( ) function may fail if:

[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

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

SYNOPSIS

UX

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

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

DESCRIPTION

The *listen*( ) function marks a connection-oriented socket, specified by the *socket* argument, as accepting connections, and limits the number of outstanding connections in the socket's listen queue to the value specified by the *backlog* argument.

If *listen*( ) is called with a *backlog* argument value that is less than 0, the function sets the length of the socket's listen queue to 0.

Implementations may limit the length of the socket's listen queue. If *backlog* exceeds the implementation-dependent maximum queue length, the length of the socket's listen queue will be set to the maximum supported value.

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

The *listen*( ) function may fail if:

[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

UX 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

UX 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

UX 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

UX

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

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

DESCRIPTION

The *recv*( ) function receives messages from a connected socket. 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, the connection is terminated, or 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 [EWOULDBLOCK] or [EAGAIN].

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:

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

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

[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-oriented 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.

[EWOULDBLOCK] or [EAGAIN]
The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.

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

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

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

DESCRIPTION

The *recvfrom*( ) function receives a message from a connection-oriented or connectionless socket. It is normally used with connectionless 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, the connection is terminated, or 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 [EWOULDBLOCK] or [EAGAIN].

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:

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

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

[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-oriented 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.

[EWOULDBLOCK] or [EAGAIN]
The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.

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

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

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

DESCRIPTION

The *recvmsg*( ) function receives a message from a connection-oriented or connectionless socket. It is normally used with connectionless 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, the connection is terminated, or 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 [EWOULDBLOCK] or [EAGAIN].

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** members describe the scatter/gather locations.

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:

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

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

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

[EWOULDBLOCK] or [EAGAIN]
The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.

[EINTR] This function was interrupted by a signal before any data was available.

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

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

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

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

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

The *recvmsg*( ) function may fail if:

[EINVAL] The **msg_iovlen** member of the **msghdr** structure pointed to by *msg* is less than or equal to 0, or is greater than {IOV_MAX}.

[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

UX 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

UX

```
#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.

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.

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:

[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-oriented and no peer address is set.
[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-oriented and the peer is closed or shut down for reading. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.

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

The *send*( ) function may fail if:

[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.

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

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

UX

```
#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-oriented or connectionless socket. If the socket is connectionless, the message will be sent to the address specified by *msghdr*. If the socket is connection-oriented, 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.

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.

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:

- [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.
- [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.
- [ENOTCONN] The socket is connection-oriented 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-oriented and the peer is closed or shut down for reading. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.

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

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

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

[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:

[EDESTADDRREQ] The socket is not connection-oriented 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 **msg_iovlen** member of the **msghdr** structure pointed to by *msg* is less than or equal to 0, or is greater than {IOV_MAX}.

[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 connection-oriented and 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

UX

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

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

DESCRIPTION

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

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.

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

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

[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-oriented 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-oriented and the peer is closed or shut down for reading. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.

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

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

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

[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:

[EDESTADDRREQ] The socket is not connection-oriented 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 connection-oriented and 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 *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

UX

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

int setsockopt(int socket, int level, int option_name, const void
               *option_value, size_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 protocol number 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 the protocol number of TCP, as defined in the <**netinet/in.h**> header, or as determined by using *getprotobyname*( ).

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 socket level options can be enabled or disabled. 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.
SO_BROADCAST	Permits sending of broadcast messages, if this is supported by the protocol. This option takes an **int** value.
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.
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.
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.
SO_SNDBUF	Sets send buffer size. This option takes an **int** value.

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

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.

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

[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

UX

```
#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.
[ENOTCONN]	The socket is not connected.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EINVAL]	The *how* argument is invalid.

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

UX

```
#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.

The **<sys/socket.h>** header defines at least the following values for the *domain* argument:

AF_UNIX	File system pathnames.
AF_INET	Internet address.

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-oriented byte streams, and may provide a transmission mechanism for out-of-band data.
SOCK_DGRAM	Provides datagrams, which are connectionless, unreliable messages of fixed maximum length.
SOCK_SEQPACKET	Provides sequenced, reliable, bidirectional, connection-oriented 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.

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:

[EACCES]	The process does not have appropriate privileges.
[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:

[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

UX

```
#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-oriented byte streams, and may provide a transmission mechanism for out-of-band data.
SOCK_DGRAM	Provides datagrams, which are connectionless, unreliable messages of fixed maximum length.
SOCK_SEQPACKET	Provides sequenced, reliable, bidirectional, connection-oriented 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.

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.

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

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

[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

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

CHANGE HISTORY

First released in Issue 4.

Chapter 9

Sockets Headers

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 8 specifies the headers that an application must include in order to use that function. In most cases only one header is required. These headers are present on an application development system; they do not have to be present on the target execution system.

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

UX 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

sys/socket.h — Internet Protocol family

SYNOPSIS

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

DESCRIPTION

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

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 **msghdr** structure that includes at least the following members:

`void`	`*msg_name`	optional address
`size_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
`size_t`	`msg_controllen`	ancillary data buffer len
`int`	`msg_flags`	flags on received message

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

`size_t`	`cmsg_len`	data byte count, including hdr
`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_DEBUG	Debugging information is being recorded.
SO_ACCEPTCONN	Socket is accepting connections.
SO_BROADCAST	Transmission of broadcast messages is supported.
SO_REUSEADDR	Reuse of local addresses is supported.
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_SNDBUF	Send buffer size.
SO_RCVBUF	Receive buffer size.
SO_ERROR	Socket error status.
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_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_INET	Internet domain sockets

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,
             size_t *address_len);
int     bind(int socket, const struct sockaddr *address,
             size_t address_len);
int     connect(int socket, const struct sockaddr *address,
             size_t address_len);
int     getpeername(int socket, struct sockaddr *address,
             size_t *address_len);
int     getsockname(int socket, struct sockaddr *address,
             size_t *address_len);
int     getsockopt(int socket, int level, int option_name,
             void *option_value, size_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, size_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, size_t dest_len);
int     setsockopt(int socket, int level, int option_name,
             const void *option_value, size_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]);
```

SEE ALSO

accept( ), *bind*( ), *connect*( ), *getpeername*( ), *getsockname*( ), *getsockopt*( ), *listen*( ), *recv*( ), *recfrom*( ), *recvmsg*( ), *send*( ), *sendmsg*( ), *sendto*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), *socketpair*( ).

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

UX 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/un.h — definitions for UNIX-domain sockets

SYNOPSIS

UX
```
#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>**.

SEE ALSO

bind( ), *socket*( ), *socketpair*( ).

Chapter 10

IP Address Resolution Interfaces

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, gethostbyaddr, gethostbyname, gethostent, sethostent — network host database functions

SYNOPSIS

UX

```
#include <netdb.h>

extern int h_errno;

void endhostent(void);

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

struct hostent *gethostbyname(const char *name);

struct hostent *gethostent(void);

void sethostent(int stayopen);
```

DESCRIPTION

The *gethostent*( ), *gethostbyaddr*( ), and *gethostbyname*( ) functions each return a pointer to a **hostent** structure, the members of which contain the fields of an entry in the network host database.

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

The *gethostbyaddr*( ) function searches the database from the beginning and finds the first entry for which the address family specified by *type* matches the **h_addrtype** member and the address pointed to by *addr* occurs in *h_addrlist*, opening a connection to the database if necessary. The *addr* argument is a pointer to the binary-format (that is, not null-terminated) address in network byte order, whose length is specified by the *len* argument. The datatype of the address depends on the address family. For an address of type AF_INET, this is an **in_addr** structure, defined in **<netinet/in.h>**.

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

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

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

RETURN VALUE

On successful completion, *gethostbyaddr*( ), *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. Otherwise, a null pointer is returned.

On unsuccessful completion, *gethostbyaddr*( ) and *gethostbyname*( ) functions set *h_errno* to indicate the error.

ERRORS

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

The *gethostbyaddr*( ) and *gethostbyname*( ) functions will fail in the following cases, setting *h_errno* to the value shown in the list below. Any changes to *errno* are unspecified.

[HOST_NOT_FOUND]
: No such host is known.

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

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

[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.

APPLICATION USAGE

The *gethostent*( ), *gethostbyaddr*( ), and *gethostbyname*( ) functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

These functions are generally used with the Internet address family.

SEE ALSO

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

CHANGE HISTORY

First released in Issue 4.

NAME

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

SYNOPSIS

UX

```
#include <netdb.h>

void endnetent(void);

struct netent *getnetbyaddr(in_addr_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.

These functions are generally used with the Internet address family.

SEE ALSO

<netdb.h>.

CHANGE HISTORY

First released in Issue 4.

NAME

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

SYNOPSIS

UX

```
#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.

These functions are generally used with the Internet address family.

SEE ALSO

<netdb.h>.

CHANGE HISTORY

First released in Issue 4.

NAME

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

SYNOPSIS

UX

```
#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.

These functions are generally used with the Internet address family.

SEE ALSO

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

CHANGE HISTORY

First released in Issue 4.

NAME

gethostbyaddr, gethostbyname, gethostent — network host database functions

SYNOPSIS

UX

```
#include <netdb.h>

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

struct hostent *gethostbyname(const char *name);

struct hostent *gethostent(void);
```

DESCRIPTION

Refer to *endhostent*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

gethostname — get name of current host

SYNOPSIS

UX

```
#include <unistd.h>

int gethostname(char *name, size_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

gethostid( ) (in the **XSH** specification), *uname*( ), **<unistd.h>**.

CHANGE HISTORY

First released in Issue 4.

NAME

getnetbyaddr, getnetbyname, getnetent — network database functions

SYNOPSIS

UX

```
#include <netdb.h>

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

struct netent *getnetbyname(const char *name);

struct netent *getnetent(void);
```

DESCRIPTION

Refer to *endnetent*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

getprotobynumber, getprotobyname, getprotoent — network protocol database functions

SYNOPSIS

UX

```
#include <netdb.h>

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void);
```

DESCRIPTION

Refer to *endprotoent*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

getservbyport, getservbyname, getservent — network services database functions

SYNOPSIS

UX

```
#include <netdb.h>

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

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

struct servent *getservent(void);
```

DESCRIPTION

Refer to *endservent*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

h_errno — error return value for network database operations

SYNOPSIS

UX
```
extern int h_errno;
```

DESCRIPTION

Refer to *endhostent*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

htonl, htons, ntohl, ntohs — convert values between host and network byte order

SYNOPSIS

UX

```
#include <arpa/inet.h>

in_addr_t htonl(in_addr_t hostlong);

in_port_t htons(in_port_t hostshort);

in_addr_t ntohl(in_addr_t netlong);

in_port_t ntohs(in_port_t netshort);
```

DESCRIPTION

These functions convert 16-bit and 32-bit quantities between network byte order and host byte order.

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 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 — Internet address manipulation

SYNOPSIS

UX

```
#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

The *inet_addr*( ) function converts the string pointed to by *cp*, in the Internet standard dot 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 Internet standard dot 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 dot 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 dot 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

ntohl, ntohs — convert values between host and network byte order

SYNOPSIS

UX

```
#include <arpa/inet.h>

in_addr_t ntohl(in_addr_t netlong);

in_port_t ntohs(in_port_t netshort);
```

DESCRIPTION

Refer to *htonl*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

sethostent — network host database function

SYNOPSIS

UX

```
#include <netdb.h>

void sethostent(int stayopen);
```

DESCRIPTION

Refer to *endhostent*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

setnetent — network database function

SYNOPSIS

UX

```
#include <netdb.h>

void setnetent(int stayopen);
```

DESCRIPTION

Refer to *endnetent*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

setprotoent — network protocol database function

SYNOPSIS

UX

```
#include <netdb.h>

void setprotoent(int stayopen);
```

DESCRIPTION

Refer to *endprotoent*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

setservent — network services database function

SYNOPSIS

UX

```
#include <netdb.h>

void setservent(int stayopen);
```

DESCRIPTION

Refer to *endservent*( ).

CHANGE HISTORY

First released in Issue 4.

Chapter 11

IP Address Resolution Headers

This chapter provides reference manual pages on the headers for the Address Resolution API.

NAME

arpa/inet.h — definitions for internet operations

SYNOPSIS

UX `#include <arpa/inet.h>`

DESCRIPTION

The <**arpa/inet.h**> header defines the type **in_port_t** and the type **in_addr_t** as defined in <**netinet/in.h**>.

The <**arpa/inet.h**> header defines the **in_addr** structure, as defined in <**netinet/in.h**>.

The following may be declared as functions, or defined as macros, or both:

```
in_addr_t      htonl(in_addr_t hostlong);
in_port_t      htons(in_port_t hostshort);
in_addr_t      ntohl(in_addr_t netlong);
in_port_t      ntohs(in_port_t netshort);
```

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);
```

Inclusion of the **<arpa/inet.h>** header may also make visible all symbols from **<netinet/in.h>**.

SEE ALSO

htonl( ), *inet_addr*( ), **<netinet/in.h>**.

CHANGE HISTORY

First released in Issue 4.

NAME

netdb.h — definitions for network database operations

SYNOPSIS

UX
```
#include <netdb.h>
```

DESCRIPTION

The **<netdb.h>** header defines the type **in_port_t** and the type **in_addr_t** as defined in **<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.
`in_addr_t n_net`	The network number, in host byte order.

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.

The **<netdb.h>** header provides a declaration for *h_errno*:

```
extern int h_errno;
```

The **<netdb.h>** header defines the following macros for use as error values for *gethostbyaddr*( ) and *gethostbyname*( ):

HOST_NOT_FOUND
NO_DATA
NO_RECOVERY
TRY_AGAIN

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);
struct hostent  *gethostbyaddr(const void *addr, size_t len, int type);
struct hostent  *gethostbyname(const char *name);
struct hostent  *gethostent(void);
struct netent   *getnetbyaddr(in_addr_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>**.

SEE ALSO

endhostent( ), *endnetent*( ), *endprotoent*( ), *endservent*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

netinet / in.h — Internet Protocol family

SYNOPSIS

UX
```
#include <netinet/in.h>
```

DESCRIPTION

The **<netinet/in.h>** header defines the following types 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 member:

```
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 Dummy for 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 Local host address.

INADDR_BROADCAST Broadcast address.

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, int address_len);
```

SEE ALSO

gethostname( ).

CHANGE HISTORY

First released in Issue 4.

ISO Transport Protocol Information

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[6].

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-oriented 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-oriented 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.

---

6. The mapping for ISO-over-TCP that is referred to in this Appendix is that defined by RFC-1006: *ISO Transport Service on top of the TCP*, Version 3, May 1987, Marshall T Rose and Dwight E Cass, Network Working Group, Northrop Research & Technology Center. See also the *X/Open Guide to IPS-OSI Coexistence and Migration*. 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 6. 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 194. 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 (see Chapter 6). 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
TCO_THROUGHPUT	struct thrpt	octets per second	throughput
TCO_TRANSDEL	struct transdel	time in milliseconds	transit delay
TCO_RESERRORRATE	struct rate	OPT_RATIO	residual error rate
TCO_TRANSFFAILPROB	struct rate	OPT_RATIO	transfer failure probability
TCO_ESTFAILPROB	struct rate	OPT_RATIO	connection establ. failure probability
TCO_RELFAILPROB	struct rate	OPT_RATIO	connection release failure probability
TCO_ESTDELAY	struct rate	time in milliseconds	connection establ. delay
TCO_RELDELAY	struct rate	time in milliseconds	connection release delay
TCO_CONNRESIL	struct rate	OPT_RATIO	connection resilience
TCO_PROTECTION	unsigned long	see text	protection
TCO_PRIORITY	unsigned long	see text	priority
TCO_EXPD	unsigned long	T_YES/T_NO	expedited data

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 (TCO_RESERRORRATE).

Absolute Requirements

For the options in Table A-1 on page 190, 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 TCO_PROTECTION is an absolute requirement if the T_ABSREQ flag is set. The values specified for TCO_EXPD and 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 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 6.
- 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 6.3.4 on page 39).
- The values of the QOS options TCO_THROUGHPUT, TCO_TRANSDEL, TCO_RESERRORRATE, TCO_TRANSFFAILPROB, TCO_ESTFAILPROB, TCO_RELFAILPROB, TCO_ESTDELAY, TCO_RELDELAY and 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 TCO_PROTECTION, TCO_PRIORITY and TCO_EXPD.
- TCO_THROUGHPUT and 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 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.
- 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.

- 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.

- An ISO-over-TCP transport provider may not support Quality of Service parameter negotiation. If not, an attempt to negotiate a Quality of Service option with an ISO-over-TCP transport provider will return with the status field set to T_NOTSUPPORT.

- 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. 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**
TCO_LTPDU	unsigned long	length in octets	maximum length of TPDU
TCO_ACKTIME	unsigned long	time in milliseconds	acknowledge time
TCO_REASTIME	unsigned long	time in seconds	reassignment time
TCO_PREFCLASS	unsigned long	see text	preferred class
TCO_ALTCLASS1	unsigned long	see text	1st alternative class
TCO_ALTCLASS2	unsigned long	see text	2nd alternative class
TCO_ALTCLASS3	unsigned long	see text	3rd alternative class
TCO_ALTCLASS4	unsigned long	see text	4th alternative class
TCO_EXTFORM	unsigned long	T_YES / T_NO / T_UNSPEC	extended format
TCO_FLOWCTRL	unsigned long	T_YES / T_NO / T_UNSPEC	flowctr
TCO_CHECKSUM	unsigned long	T_YES / T_NO / T_UNSPEC	checksum
TCO_NETEXP	unsigned long	T_YES / T_NO / T_UNSPEC	network expedited data
TCO_NETRECPTCF	unsigned long	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 6).
- 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 6.3.4 on page 39).
- 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 192. 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, 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 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 powers of 2 between 2**7 and 2**13.

  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 190. 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 6). 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 TCO_EXP is set to T_YES and TCO_PREFCLASS is set to T_CLASS2, TCO_FLOWCTRL must also be set to T_YES.

- If TCO_PREFCLASS is set to T_CLASS0, TCO_EXP must be set to T_NO.
- The value in TCO_PREFCLASS must not be lower than the value in TCO_ALTCLASS1, 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 190).

All options are association-related (see Chapter 6). 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
TCL_TRANSDEL	struct rate	time in milliseconds	transit delay
TCL_RESERRORRATE	struct rate	OPT_RATIO	residual error rate
TCL_PROTECTION	unsigned long	see text	protection
TCL_PRIORITY	unsigned long	see text	priority

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 TCO_RESERRORRATE and TCL_RESERRORRATE, TCO_PRIORITY and TCL_PRIORITY, TCO_PROTECTION and TCL_PROTECTION (see Table A-1 on page 190, ISO 8072:1986).
- TCL_TRANSDEL and TCO_TRANSDEL are different. 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 TCO_TRANSDEL. The range of legal option values for each field of **struct rate** is the same as that of 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
TCL_CHECKSUM	unsigned long	T_YES/T_NO	checksum computation

Table A-4 Management Option

Absolute Requirements

A request for this option is an absolute requirement.

Further Remarks

TCL_CHECKSUM is the option allows disabling/enabling of the checksum computation. The legal values are T_YES (checksum enabled) and T_NO (checksum disabled).

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_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 –2 if no expedited data transfer was negotiated, and to 16 otherwise. The remaining fields are set according to the characteristics of the transport protocol class in use for this connection, as defined in the table below.

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 (1)	x (1)	x (1)	x (1)
info->tsdu	/	x (2)	x (2)	0->63488	x (2)
info->etsdu	/	−2	16 / −2 (3)	−2	16 / −2
info->connect	/	−2	32	−2	32 / −2
info->discon	/	−2	64	−2	64 / −2
info->servtype	/	T_COTS	T_COTS	T_CLTS	T_COTS
info->flags	/	0	0	0	0

1. ‘x’ equals −2 or an integral number greater than zero.
2. ‘x’ equals -1 or an integral number greater than zero.
3. Depending 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 connect indication, *call->udata.maxlen* should be set to 32 before the call to *t_listen*( ).

If the user has set *qlen* greater than 1 (on the call to *t_bind*( )), the user may queue up several connect indications before responding to any of them. The user should be forewarned that the ISO transport provider may start a timer to be sure of obtaining a response to the connect request in a finite time. So if the user queues the connect 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 table below 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 (1)	x (1)	x (1)	x (1)
info->tsdu	/	x (2)	x (2)	0->63488	x (2)
info->etsdu	/	–2	16	–2	16 / –2
info->connect	/	–2	32	–2	32 / –2
info->discon	/	–2	64	–2	64 / –2
info->servtype	/	T_COTS	T_COTS	T_CLTS	T_COTS
info->flags	/	0	0	0	0

1. ‘x’ equals –2 or an integral number greater than zero.
2. ‘x’ equals –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 transport service does not support the sending of expedited data. This function is not supported by an ISO-over-TCP transport provider.

Internet Protocol-specific Information

B.1 General

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

Notes

- T_MORE flag and TSDUs

  The notion of TSDU is not supported by a TCP transport provider, so the T_MORE flag will be ignored when TCP is used. The TCP PUSH flag cannot be used through the XTI interface because the TCP Military Standard (see Referenced Documents) states that:

  ‘‘Successive pushes may not be preserved because two or more units of pushed data may be joined into a single pushed unit by either the sending or receiving TCP. Pushes are not visible to the receiving Upper Level Protocol and are not intended to serve as a record boundary marker’’.

- Expedited data

  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. UDP has no urgent mechanism. See the TCP Military 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. However, its use is not recommended so that applications using TCP may be ported to use ISO Transport. 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.

- Connection establishment

  TCP does not allow the possibility of refusing a connection indication. Each connect 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.

B.2 Options

Options are formatted according to the structure **t_opthdr** as described in Chapter 6. A transport provider compliant to this specification supports none, all or any subset of the options defined in Section B.2.1, Section B.2.2 and Section B.2.3. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode.

B.2.1 TCP-level Options

The protocol level is INET_TCP. For this level, Table B-1 shows the options that are defined.

Option Name	Type of Option Value	Legal Option Value	Meaning
TCP_KEEPALIVE	struct t_kpalive	see text	check if connections are alive
TCP_MAXSEG	unsigned long	length in octets	get TCP maximum segment size
TCP_NODELAY	unsigned long	T_YES / T_NO	don't delay send to coalesce packets

Table B-1 TCP-level Options

These options are *not* association-related. They may be negotiated in all XTI states except T_UNBND and T_UNINIT. They are read-only in state T_UNBND. See Chapter 6 for the difference between options that are association-related and those that are not.

Absolute Requirements

A request for TCP_NODELAY and a request to activate TCP_KEEPALIVE is an absolute requirement. TCP_MAXSEG is a read-only option.

Further Remarks

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:

‘‘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 {
    long kp_onoff;    /* switch option on/off           */
    long 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
T_YES | T_GARBAGE	activate keep-alive timer and send garbage octet

Usually, an implementation should send a keep-alive packet with no data (T_GARBAGE not set). If T_GARBAGE is set, the keep-alive packet contains one garbage octet for compatibility with erroneous TCP implementations.

An implementation is, however, not obliged to support T_GARBAGE (see RFC 1122). Since the *kp_onoff* value is an absolute requirement, the request ‘‘T_YES | T_GARBAGE’’ may therefore be rejected.

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 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.

TCP_MAXSEG
: This option is read-only. It is used to retrieve the maximum TCP segment size.

TCP_NODELAY
: Under most circumstances, TCP sends data as soon as it is presented. When outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgement is received. For a small number of clients, such as window systems (for example, MIT X Window System) that send a stream of mouse events which receive no replies, this packetisation may cause significant delays. TCP_NODELAY is used to defeat this algorithm. Legal option values are T_YES (‘‘don’t delay’’) and T_NO (‘‘delay’’).

B.2.2 UDP-level Options

The protocol level is INET_UDP. The option defined for this level is shown in Table B-2.

Option Name	Type of Option Value	Legal Option Value	Meaning
UDP_CHECKSUM	unsigned long	T_YES / T_NO	checksum computation

Table B-2 UDP-level Option

This option is association-related. It may be negotiated in all XTI states except T_UNBND and T_UNINIT. It is read-only in state T_UNBND. See Chapter 6 for the difference between options that are association-related and those that are not.

Absolute Requirements

A request for this option is an absolute requirement.

Further Remarks

UDP_CHECKSUM — The option allows disabling/enabling of the 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.

B.2.3 IP-level Options

The protocol level is INET_IP. The options defined for this level are listed in Table B-3.

Option Name	Type of Option Value	Legal Option Value	Meaning
IP_BROADCAST	unsigned int	T_YES/T_NO	permit sending of broadcast messages
IP_DONTROUTE	unsigned int	T_YES/T_NO	just use interface addresses
IP_OPTIONS	array of unsigned characters	see text	IP per-packet options
IP_REUSEADDR	unsigned int	T_YES/T_NO	allow local address reuse
IP_TOS	unsigned char	see text	IP per-packet type of service
IP_TTL	unsigned char	time in seconds	IP per packet time-to-live

Table B-3 IP-level Options

IP_OPTIONS and IP_TOS are both association-related options. All other options are *not* association-related. See Chapter 6 for the difference between association-related options and options that are not.

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

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.

IP_DONTROUTE — This option indicates that outgoing messages should bypass the standard routing facilities. It is mainly used for testing and development.

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 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 IP_OPTIONS that is sent with outgoing datagrams.

Common applications never need this option. It is mainly used for network debugging and control purposes.

IP_REUSEADDR — Many TCP implementations do not allow the user to bind more than one transport endpoint to addresses with identical port numbers. If 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.

IP_TOS — This option is used to set (retrieve) the *type-of-service* field of an outgoing (incoming) 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 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 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.

IP_TTL — This option 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 association-related option).

B.3 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. Also, *resfd* must be bound to the same address as *fd*. A potential restriction on binding of endpoints to protocol addresses is described under *t_bind*( ) below.

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

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

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**	
		TCP/IP	**UDP/IP**
name	x	/	/
oflag	x	/	/
info->addr	/	x	x
info->options	/	x	x
info->tsdu	/	0	x
info->etsdu	/	–1	–2
info->connect	/	–2	–2
info->discon	/	–2	–2
info->servtype	/	T_COTS / T_COTS_ORD	T_CLTS
info->flags	/	T_SNDZERO	T_SNDZERO

‘x’ equals -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 remote calling socket.

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 TSDU varies among implementations.

Appendix C

Guidelines for Use of XTI

C.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 5-5 on page 32, *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 must be bound to an appropriate protocol address and must be in the T_IDLE state.
  - — The user transferring the current state of an endpoint must have correct permissions for the use of the protocol address bound to the accepting transport endpoint.
  - — The endpoint that transfers the state of the transport interface is placed into the T_IDLE state by the transport provider after the completion of the transfer if there are no more outstanding connect indications.

C.2 Example in Connection-oriented Mode

Table C-1 on page 209 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 connect request (via the return from *t_listen*( )) and then would call the *t_accept*( ).

The state diagram in Table C-1 on page 209 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 5-7 on page 33.

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

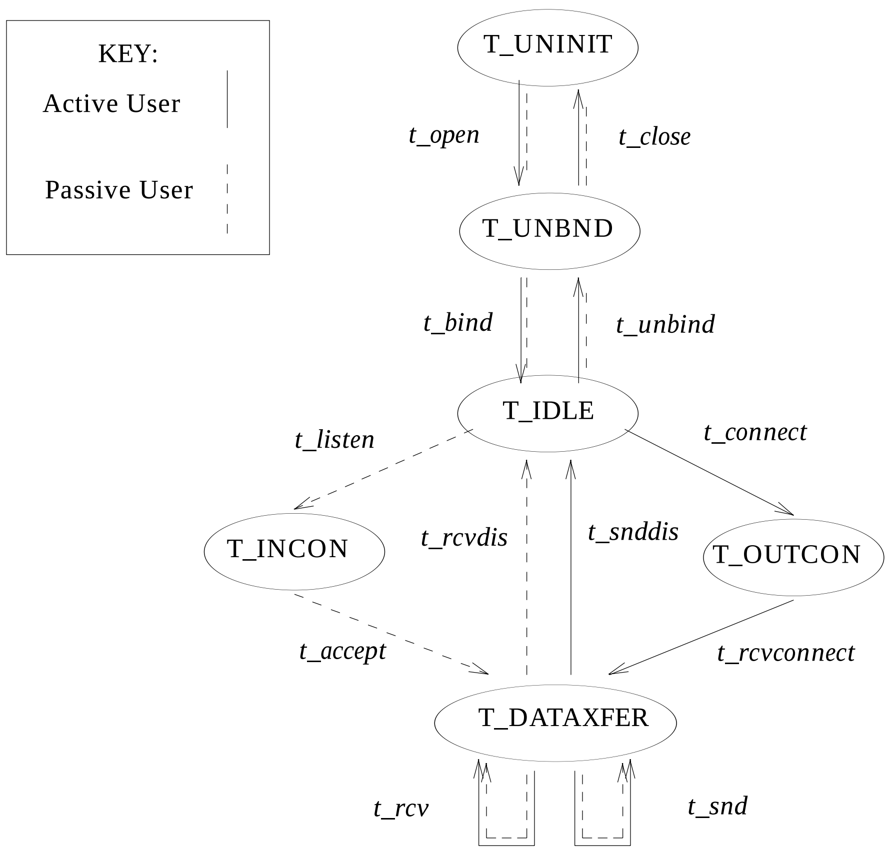


Table C-1 Sequence of Transport Functions in Connection-oriented Mode

C.3 Example in Connectionless Mode

Table C-2 shows the allowable sequence of functions of user A and user B communicating using a connectionless transport service. This example is not meant to show all the functions that must be called but rather to highlight the important functions that request a particular service. Blank lines are used to indicate that a function would be called by another user prior to a related function being called by the remote user.

The state diagram that follows shows the flow of the events through the various states. This example shows a successful exchange of data between user A and user B. For a detailed description of all possible states and events, see Table 5-7 on page 33.

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

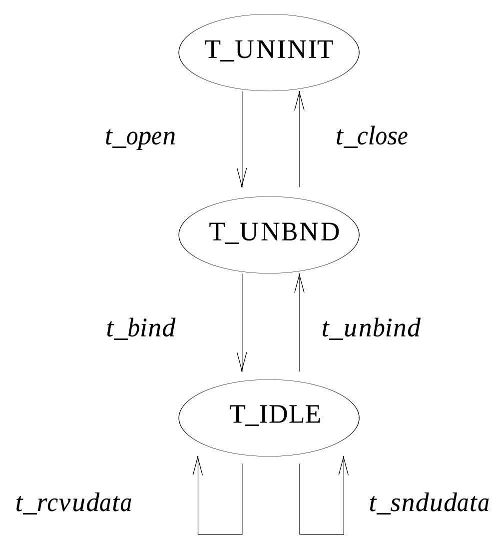


Table C-2 Sequence of Transport Functions in Connectionless Mode

C.4 Writing Protocol-independent Software

In order to maximise portability of XTI applications between different kinds of machine 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 connect establishment function. Optionally, the user can choose not to pass options as an argument on connect establishment functions.

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

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

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

9. The optional orderly release facility of the 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.

C.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 3.7 on page 14 and Section 3.8 on page 16.

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.

C.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.

C.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	Disconnect 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.

C.6 The Poll Function

poll( ) is defined in the System V Interface Definition, Third Edition as follows. Note that this definition may vary slightly in other systems.

UX If the implementation defines _XOPEN_UNIX, refer to the description of *poll*( ) in the **XSH** specification. Moreover, Chapter 8 on page 105 of the current document gives additional information on the specific effect of *poll*( ) when applied to Sockets.

The manual page definition on the next page is followed by a section giving guidelines for use of System V *poll*( ).

NAME

poll - input/output multiplexing

SYNOPSIS

```
#include <poll.h>
int poll(struct pollfd fds[], unsigned long nfds, int timeout);
```

DESCRIPTION

poll( ) provides users with a mechanism for multiplexing input/output over a set of file descriptors. *poll*( ) identifies those file descriptors on which a user can read or write data, or on which certain events have occurred. A user can read data using *read*( ) and write data using *write*( ). For STREAMS file descriptors, a user can also receive messages using *getmsg*( ) and *getpmsg*( ), and send messages using *putmsg*( ) and *putpmsg*( ).

fds specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are *pollfd* structures which contain the following members:

```
int fd;           /* file descriptor  */
short events;     /* requested events */
short revents;    /* returned events  */
```

where *fd* specifies an open file descriptor and *events* and *revents* are bit-masks constructed by OR'ing a combination of the following event flags:

POLLIN	Data other than high-priority data may be read without blocking. For STREAMS, this flag is set even if the message is of zero length.
POLLRDNORM	Normal data (priority band equals 0) may be read without blocking. For STREAMS, this flag is set even if the message is of zero length.
POLLRDBAND	Data from a non-zero priority band may be read without blocking. For STREAMS, this flag is set even if the message is of zero length.
POLLPRI	High-priority data may be received without blocking. For STREAMS, this flag is set even if the message is of zero length.
POLLOUT	Normal data may be written without blocking.
POLLWRBAND	Priority data (priority band greater than 0) may be written.
POLLER	An error has occurred on the device or STREAM. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.
POLLUP	The device has been disconnected. This event and POLLOUT are mutually exclusive; a STREAM can never be writable if a hangup has occurred. However, this event and POLLIN, POLLRDNORM, POLLRDBAND or POLLPRI are not mutually exclusive. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.
POLLNVAL	The specified *fd* value is invalid. This flag is only valid in the *revents* field; it is not used in the *events* field.

For each element of the array pointed to by *fds*, *poll*( ) examines the given file descriptor for the event(s) specified in *events*. The number of file descriptors to be examined is specified by *nfds*.

If the value of *fd* is less than zero, *events* is ignored and *revents* is set to zero in that entry on return from *poll*( ).

The results of the *poll*( ) query are stored in the *revents* field in the *pollfd* structure. Bits are set in the *revents* bitmask to indicate which of the requested events are true. If none of the requested events are true, none of the specified bits is set in *revents* when the *poll*( ) call returns. The events flags POLLUP, POLLERR and POLLNVAL, are always set in the *revents* if the conditions they indicate are true; this occurs even though these flags were not present in *events*.

If none of the defined events have occurred on any selected file descriptor, *poll*( ) waits at least *timeout* milliseconds for an event to occur on any of the selected file descriptors. On a computer where millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. If the value of *timeout* is 0, *poll*( ) returns immediately. If the value of *timeout* is −1 *poll*( ) blocks until a requested event occurs or until the call is interrupted. *poll*( ) is not affected by the O_NDELAY and O_NONBLOCK flags.

RETURN VALUES

Upon successful completion, the function *poll*( ) returns a non-negative value. A positive value indicates the total number of file descriptors that have been selected (that is, file descriptors for which the *revents* field is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, the function *poll*( ) returns a value −1 and sets *errno* to indicate an error.

ERRORS

Under the following conditions, the function *poll*( ) fails and sets *errno* to:

EAGAIN	If the allocation of internal data structures failed but the request should be attempted again.
EINTR	If a signal was caught during the *poll*( ) system call.
EINVAL	If the argument *nfds* is less than zero or greater than {OPEN_MAX}.

C.7 Use of Poll

For an application to be notified of any XTI events on each of its active endpoints, the array pointed to by *fds* should contain as many elements as active endpoints identified by the file descriptor *fd*, and the *events* member of those elements should be set to the combination of event flags as specified below:

- For Class 1 events:

  POLLIN | POLLPRI (for System V Release 3)

  or:

  POLLIN | POLLRDNORM | POLLRDBAND | POLLPRI (for System V Release 4).

- For Class 2 events:

  POLLOUT (for System V Release 3)

  or:

  POLLOUT | POLLWRBAND (for System V Release 4).

In a System V Release 3, the meaning of POLLOUT may differ for different XTI implementations. It could either mean:

- that both normal and expedited data may be sent

  or:

- that normal data may be sent and the flow of expedited data cannot be monitored via *poll*( ).

A truly portable XTI application should, therefore, not assume that the flow of expedited data is monitored by *poll*( ). This is not a serious restriction, since an application usually only sends small amounts of expedited data and flow restrictions are not a major problem.

In a System V Release 4, the meaning of POLLOUT and POLLWRBAND is intended to be the same for all XTI implementations.

POLLOUT Normal data may be sent.

POLLWRBAND Expedited data may be sent.

The following description gives the outline of an XTI server program making use of the System V *poll*( ).

```
/*
 * 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.
 * The number of poll flags has increased from System V Release 3 to
 * System V Release 4. Hence, if this program is to be used in a
 * System V Release 3, the constant SVR3 must be defined during
 * compile time.
 *
 * 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 connect indications received, a new endpoint is opened
 * with qlen = 0 and the connect 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 disconnect 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 <stropts.h>
#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.
 */

#ifdef SVR3
#define ERREVENTS    (POLLERR | POLLHUP | POLLNVAL)
#define INEVENTS     (POLLIN | POLLPRI)
#define OUTEVENTS    POLLOUT
#else
#define ERREVENTS    (POLLERR | POLLHUP | POLLNVAL)
#define INEVENTS     (POLLIN | POLLRDNORM | POLLRDBAND | POLLPRI)
#define OUTEVENTS    (POLLOUT | POLLWRBAND)
#endif
#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 connect 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 connect 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 disconnect 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] Disconnect 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 disconnect 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);
}
```

C.8 The Select Function

select( ) is defined in the 4.3 Berkeley Software Distribution as follows. Note that this definition may vary slightly in other systems.

UX If the implementation defines _XOPEN_UNIX, refer to the description of *select*( ) in the **XSH** specification. Moreover, Chapter 8 on page 105 of the current document gives additional information on the specific effect of *select*( ) when applied to Sockets.

The manual page for this definition is given on the next page, and this is followed by as section giving guidelines for Use of BSD *select*( ).

NAME

select - synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>

nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

select( ) examines the I/O descriptor sets whose addresses are passed in *readfds, writefds* and *exceptfds* to see if some of their descriptors are ready for reading, ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; that is, the descriptors from 0 through *nfds* -1 in the descriptor sets are examined. On return, *select*( ) replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: *FD_ZERO(&fdset)* initialises a descriptor set *fdset* to the null set. *FD_SET(fd, &fdset)* includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd, &fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, &fdset)* is non-zero if *fd* is a member of *fdset*, zero otherwise. The behaviour of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To affect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued *timeval* structure.

Any of *readfds, writefds* and *exceptfds* may be given as zero pointers if no descriptors are of interest.

RETURN VALUES

select( ) returns the number of ready descriptors that are contained in the descriptor sets, or –1 if an error occurred. If the time limit expires then *select*( ) returns 0. If *select*( ) returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from *select*( ) indicates:

[EBADF] One of the descriptor sets specified an invalid descriptor.

[EINTR] A signal was delivered before the time limit expired and before any of the selected events occurred.

[EINVAL] The specified time limit is invalid. One of its components is negative or too large.

C.9 Use of Select

Many systems provide the macros *FD_SET*, *FD_CLR*, *FD_ISSET* and *FD_ZERO* in **<sys/types.h>** or other header files to manipulate these bit masks. If not available they should be defined by the user (see the program example below).

For an application to be notified of any XTI events on each of its active endpoints identified by a file descriptor *fd*, this file descriptor *fd* should be included in the appropriate descriptor sets *readfds*, *exceptfds* or *writefds* as specified below:

- For Class 1 events:

  Set the bit masks *readfds* and *exceptfds* by *FD_SET(fd, readfds)* and *FD_SET(fd, exceptfds)*.

- For Class 2 events:

  Set the bit mask *writefds* by *FD_SET(fd, writefds)*.

If, on return of *select*( ), the bit corresponding to *fd* is set in *writefds*, this can have a different meaning for different XTI implementations. It could either mean:

- that both normal and expedited data may be sent

  or:

- that normal data may be sent and the flow of expedited data cannot be monitored via *select*( ).

A truly portable XTI application should, therefore, not assume that the flow of expedited data is monitored by *select*( ). This is not a serious restriction, since an application usually only sends small amounts of expedited data and flow restrictions are not a major problem.

The remainder of this section describes the outline of an XTI server program making use of the BSD *select*( ).

```
/*
 * 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 connect indications received, a new endpoint is opened with
 * qlen = 0 and the connect 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 disconnect 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( ). Some UNIX derivatives use other includes,
 * for example, <sys/times.h> instead of <sys/time.h>.
 *             <sys/select.h> instead of <sys/types.h>.
 */
#include <sys/types.h>
#include <time.h>

/*
 * Includes that are only relevant, if the type fd_set and the macros
 * FD_SET, FD_CLR, FD_ISSET and FD_ZERO have to be explicitly defined
 * in this program.
 */
#include <limits.h>
#include <string.h>     /* for memset( ) */

/*
 * 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 */

/*
 * Select uses bit masks of file descriptors in longs. Most systems
 * provide a type "fd_set" and macros in <sys/types.h> or <sys/select.h>
 * to ease the use of select( ).
 * They are explicitly defined below in case that they are not defined in
 * <sys/types.h> or <sys/select.h>.
 */
/*
 * OPEN_MAX should be >= number of maximum open files per process
 */
#ifndef OPEN_MAX
#define OPEN_MAX        256
#endif
#ifndef NFDBITS
#define NFDBITS (sizeof(long) * CHAR_BIT)       /* bits per mask */
#endif
#ifndef howmany
#define howmany(x, y)   (((x)+((y)-1))/(y))
#endif
#ifndef FD_SET
typedef struct  fd_set {
        long    fds_bits[howmany(OPEN_MAX, NFDBITS)];
} fd_set;
#define FD_SET(n, p)   ((p)->fds_bits[(n)/NFDBITS]  |= (1 << ((n) % NFDBITS)))
#define FD_CLR(n, p)   ((p)->fds_bits[(n)/NFDBITS]  &= ~(1 << ((n) % NFDBITS)))
#define FD_ISSET(n, p) ((p)->fds_bits[(n)/NFDBITS]  & (1 << ((n) % NFDBITS)))
#define FD_ZERO(p)     memset(*(p), (u_char) 0, sizeof(*(p)))
#endif /* ! FD_SET */

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 connect 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 connect 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 disconnect 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] Disconnect 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 disconnect 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 D

Use of XTI to Access NetBIOS

D.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.[7].

The following CAE Specification extends that work to provide a standard programming interface to NetBIOS transport providers in X/Open-compliant systems, using an existing X/Open Common Applications Environment (CAE) interface, XTI.

The X/Open Transport Interface (XTI) defines a transport service interface that is independent of any specific transport provider.

This CAE Specification 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’’.

D.2 Objectives

The objectives of this standardisation are:

1. to facilitate the development and portability of CAE 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 CAE Specification provides a migration step to users moving from proprietary systems in a NetBIOS environment to open systems, that is, the X/Open CAE.

---

7. 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.

D.3 Scope

No extensions to XTI, as it is defined in the main body of this CAE Specification, are made in this NetBIOS CAE Specification. This NetBIOS CAE Specification is concerned only with standardisation of the mapping of XTI to the NetBIOS facilities, and not a new definition of XTI itself.

This CAE 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.

D.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.

D.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 or group names, and must be identified as such in certain circumstances.

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 X/Open CAE.

The following definitions are supplied with any implementation of XTI on top of NetBIOS. They should be included in **<xti.h>**.

```
#define NB_UNIQUE      0
#define NB_GROUP       1
#define NB_NAMELEN    16
#define 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 NB_UNIQUE or NB_GROUP.
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 2*t_listen*( ) and *t_connect*( )) this event is indicated by setting *t_errno* to [TBADF].

D.6 NetBIOS Connection Release

Native NetBIOS implementations provide a linger-on-close release mechanism whereby a transport disconnect 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 D.8 on page 245).

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_DISIN 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.

D.7 Options

No NetBIOS-specific options are defined. An implementation may, however, provide XTI-level options (see *t_optmgmt*( ) on page 76.

D.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-oriented transport endpoints. The *t_accept*( ) function will fail if a user attempts to accept a connection request on a connectionless 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 D.5 on page 242. 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 *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 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 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 D.5 on page 242. 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-oriented transport endpoints. The *t_connect*( ) function will fail if a user attempts to initiate a connection on a connectionless 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 CAE system name table (see Section D.5 on page 242.

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 D.5 on page 242.

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-oriented endpoint it is a positive value, not larger than 131070. If the *fd* is associated with a connectionless endpoint it is a positive value not larger than 65535[8].

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-oriented endpoint, or *T_CLTS* if associated with a connectionless endpoint. Optionally, may be set to *T_COTS_ORD* if the *fd* is associated with a connection-oriented endpoint and the transport provider supports the use of *t_sndrel*( ) / *t_rcvrel*( ) as described in Section D.6 on page 243.

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 D.5 on page 242; 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 D.5 on page 242.

No user data may be transferred during connection establishment (call->udata.len=0 on return).

This function may only be used with connection-oriented transport endpoints. The *t_listen*( ) function will fail if a user attempts to *listen* on a connectionless 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 CAE system name table, as may occur as a result of the NetBIOS name conflict resolution process (for example,

---

8. For the mappings to OSI and IPS protocols, the value cannot exceed 512 or 1064 respectively.

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 63.

t_optmgmt( ) No special considerations for NetBIOS transport providers.

t_rcv( ) This function may only be used with connection-oriented transport endpoints. The *t_rcv*( ) function will fail if a user attempts a receive on a connectionless 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 D.5 on page 242.

No user data may be returned to the caller (call->udata.len=0 on return).

This function may only be used with connection-oriented transport endpoints. The *t_rcvconnect*( ) function will fail if a user attempts to establish a connection on a connectionless endpoint and *t_errno* will be set to [TNOTSUPPORT].

t_rcvdis( ) The following disconnect reason codes are valid for any implementation of a NetBIOS provider under XTI:

```
#define NB_ABORT     0x18  /* session ended abnormally */
#define NB_CLOSED    0x0A  /* session closed           */
#define NB_NOANSWER  0x14  /* no answer (cannot find   */
                           /* name called              */
#define NB_OPREJ     0x12  /* session open rejected    */
```

These definitions should be included in **<xti.h>**.

t_rcvrel( ) As described in Section D.6 on page 243, a T_ORDREL event will never occur in the T_DATAXFER state, but only in the T_OUTREL state. A transport user thus has only to prepare for a call to *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 D.5 on page 242.

The *fd* associated with the *t_rcvudata*( ) function must refer to a connectionless transport endpoint. The function will fail if a user attempts to receive on a connection-oriented 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 CAE 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 NB_BCAST_NAME.

t_rcvuderr( ) — If attempted on a connectionless 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-oriented 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-oriented transport endpoints. The *t_snd*( ) function will fail if a user attempts a send on a connectionless 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 93 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 D.6 on page 243 for further details.

No user data may be sent in the disconnect request (call->udata.len=0).

This function may only be used with connection-oriented transport endpoints. The *t_snddis*( ) function will fail if a user attempts a disconnect request on a connectionless 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 T_DISIN and an appropriate disconnect reason code. See Section D.6 on page 243 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 D.5 on page 242.

The *fd* associated with the *t_sndudata*( ) function must refer to a connectionless transport endpoint. The function will fail if a user attempts this function on a connection-oriented 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 CAE 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 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.

Appendix E

XTI and TLI

XTI is based on the 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.

E.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.

E.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 F

Headers and Definitions for XTI

Section 7.1 on page 47 contains a normative requirement that the contents and structures found in this appendix appear in the **<xti.h>** header.

```
/*
 * 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  /* connect confirmation received */
#define T_DATA         0x0004  /* normal data received */
#define T_EXDATA       0x0008  /* expedited data received */
#define T_DISCONNECT   0x0010  /* disconnect 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_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;

/* XTI LIBRARY FUNCTIONS */
```

UX

```
#ifndef _XOPEN_SOURCE_EXTENDED
/* XTI Library Function: t_accept - accept a connect request*/
extern int t_accept( );
/* XTI Library Function: t_alloc - allocate a library structure*/
extern char *t_alloc( );
/* XTI Library Function: t_bind - bind an address to a transport endpoint*/
extern int t_bind( );
/* XTI Library Function: t_close - close a transport endpoint*/
extern int t_close( );
/* XTI Library Function: t_connect - establish a connection */
extern int t_connect( );
/* XTI Library Function: t_error - produce error message*/
extern int t_error( );
/* XTI Library Function: t_free - free a library structure*/
extern int t_free( );
/* XTI Library Function: t_getprotaddr - get protocol addresses*/
extern int t_getprotaddr( );
/* XTI Library Function: t_getinfo - get protocol-specific service */
                                                      /* information*/
extern int t_getinfo( );
```

```
/* XTI Library Function: t_getstate - get the current state*/
extern int t_getstate( );
/* XTI Library Function: t_listen - listen for a connect indication*/
extern int t_listen( );
/* XTI Library Function: t_look - look at current event on a transport */
                                                              /* endpoint*/
extern int t_look( );
/* XTI Library Function: t_open - establish a transport endpoint*/
extern int t_open( );
/* XTI Library Function: t_optmgmt - manage options for a transport */
                                                           /* endpoint*/
extern int t_optmgmt( );
/* XTI Library Function: t_rcv - receive data or expedited data on a */
                                                          /* connection*/
extern int t_rcv( );
/* XTI Library Function: t_rcvdis - retrieve information from disconnect*/
extern int t_rcvdis( );
/* XTI Library Function: t_rcvrel - acknowledge receipt of */
/* an orderly release indication */
extern int t_rcvrel( );
/* XTI Library Function: t_rcvudata - receive a data unit*/
extern int t_rcvudata( );
/* XTI Library Function: t_rcvuderr - receive a unit data error indication*/
extern int t_rcvuderr( );
/* XTI Library Function: t_snd - send data or expedited data over a */
                                                        /* connection */
extern int t_snd( );
/* XTI Library Function: t_snddis - send user-initiated disconnect request*/
extern int t_snddis( );
/* XTI Library Function: t_sndrel - initiate an orderly release*/
extern int t_sndrel( );
/* XTI Library Function: t_sndudata - send a data unit*/
extern int t_sndudata( );
/* XTI Library Function: t_strerror - generate error message string */
extern char *t_strerror( );
/* XTI Library Function: t_sync - synchronise transport library*/
extern int t_sync( );
/* XTI Library Function: t_unbind - disable a transport endpoint*/
extern int t_unbind( );
#else
extern int   t_accept(int, int, struct t_call *);
extern char *t_alloc(int, int, int);
extern int   t_bind(int, struct t_bind *, struct t_bind *);
extern int   t_close(int);
extern int   t_connect(int, struct t_call *, struct t_call *);
extern int   t_error(char *);
extern int   t_free(char *, 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(char *, int, struct t_info *);
extern int   t_optmgmt(int, struct t_optmgmt *, struct t_optmgmt *);
extern int   t_rcv(int, char *, 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_rcvudata(int, struct t_unitdata *, int *);
```

```
extern int   t_rcvuderr(int, struct t_uderr *);
extern char *t_strerror(int);
extern int   t_snd(int, char *, unsigned int , int);
extern int   t_snddis(int, struct t_call *);
extern int   t_sndrel(int);
extern int   t_sndudata(int, struct t_unitdata *);
extern int   t_sync(int);
extern int   t_unbind(int);
#endif

/*
 * Protocol-specific service limits.
 */
struct t_info {
   long  addr;     /* max size of the transport protocol address          */
   long  options;  /* max number of bytes of protocol-specific options    */
   long  tsdu;     /* max size of a transport service data unit           */
   long  etsdu;    /* max size of expedited transport service data unit   */
   long  connect;  /* max amount of data allowed on connection            */
                   /* establishment functions                             */
   long  discon;   /* max data allowed on t_snddis and t_rcvdis functions */
   long  servtype; /* service type supported by transport provider        */
   long  flags;    /* other info about the transport provider             */
};

/*
 * Service type defines.
 */

#define T_COTS       01   /* connection-oriented transport service */
#define T_COTS_ORD   02   /* connection-oriented with orderly release */
#define T_CLTS       03   /* connectionless transport service */

/*
 * Flags defines (other info about the transport provider).
 */

#define T_SENDZERO 0x001   /* supports 0-length TSDUs */

/*
 * netbuf structure.
 */

struct netbuf {
     unsigned int   maxlen;
     unsigned int   len;
     char           *buf;
};
```

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

/*
 * t_bind - format of the address and options arguments of bind.
 */

struct t_bind {
     struct netbuf   addr;
     unsigned        qlen;
};

/*
 * Options management structure.
 */

struct t_optmgmt {
     struct netbuf   opt;
     long            flags;
};

/*
 * Disconnect 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 */
     long            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_UNUSED         -1
#define   T_NULL            0
#define   T_ABSREQ     0x8000
#define   T_INFINITE       -1
#define   T_INVALID        -2

/* T_INFINITE and T_INVALID are values of t_info */

/*
* General definitions for option management
*/
#define T_UNSPEC    (~0 - 2)  /* applicable to u_long, long, char .. */
#define T_ALLOPT    0
#define T_ALIGN(p) (((unsigned long)(p) + (sizeof (long) - 1)) \
                                        & ~(sizeof (long) - 1))
#define OPT_NEXTHDR( pbuf, buflen, popt) \
          (((char *)(popt) + T_ALIGN( (popt)->len ) < \
          (char *)(pbuf) + (buflen)) ? \
          (struct t_opthdr *)((char *)(popt) + T_ALIGN( (popt)->len )) : \
          (struct t_opthdr *)0 )


             /* 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 {
     long   l_onoff;     /* option on/off */
     long   l_linger;    /* linger time */
};

        /* 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

/*
 * Default value for the length of TPDUs.
 */

#define   T_LTPDUDFLT   128   /* define obsolete in XPG4 */

/*
 * rate structure.
 */
struct rate {
     long targetvalue;       /* target value */
     long 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 */
};

/*
 * Protocol Levels
 */

#define   ISO_TP   0x0100

/*
 * Options for Quality of Service and Expedited Data (ISO 8072:1986)
 */

#define   TCO_THROUGHPUT            0x0001
#define   TCO_TRANSDEL              0x0002
#define   TCO_RESERRORRATE          0x0003
#define   TCO_TRANSFFAILPROB        0x0004
#define   TCO_ESTFAILPROB           0x0005
#define   TCO_RELFAILPROB           0x0006
#define   TCO_ESTDELAY              0x0007
#define   TCO_RELDELAY              0x0008
#define   TCO_CONNRESIL             0x0009
#define   TCO_PROTECTION            0x000a
#define   TCO_PRIORITY              0x000b
#define   TCO_EXPD                  0x000c

#define   TCL_TRANSDEL              0x000d
#define   TCL_RESERRORRATE          TCO_RESERRORRATE
#define   TCL_PROTECTION            TCO_PROTECTION
#define   TCL_PRIORITY              TCO_PRIORITY
```

```
/*
 * Management Options
 */

#define   TCO_LTPDU              0x0100
#define   TCO_ACKTIME            0x0200
#define   TCO_REASTIME           0x0300
#define   TCO_EXTFORM            0x0400
#define   TCO_FLOWCTRL           0x0500
#define   TCO_CHECKSUM           0x0600
#define   TCO_NETEXP             0x0700
#define   TCO_NETRECPTCF         0x0800
#define   TCO_PREFCLASS          0x0900
#define   TCO_ALTCLASS1          0x0a00
#define   TCO_ALTCLASS2          0x0b00
#define   TCO_ALTCLASS3          0x0c00
#define   TCO_ALTCLASS4          0x0d00

#define   TCL_CHECKSUM           TCO_CHECKSUM

             /* INTERNET SPECIFIC ENVIRONMENT */

/*
 * TCP level
 */

#define   INET_TCP   0x6

/*
 *TCP-level Options
 */

#define   TCP_NODELAY     0x1   /* don't delay packets to coalesce */
#define   TCP_MAXSEG      0x2   /* get maximum segment size */
#define   TCP_KEEPALIVE   0x8   /* check, if connections are alive */

/*
 * Structure used with TCP_KEEPALIVE option.
 */
struct t_kpalive {
     long    kp_onoff;      /* option on/off */
     long    kp_timeout;    /* timeout in minutes */
};

#define   T_GARBAGE   0x02

/*
 * UDP level
 */

#define   INET_UDP   0x11

/*
 * UDP-level Options
 */
```

```
#define   UDP_CHECKSUM   TCO_CHECKSUM   /* checksum computation */

/*
 * IP level
 */

#define   INET_IP   0x0

/*
 * IP-level Options
 */

#define   IP_OPTIONS      0x1   /* IP per-packet options */
#define   IP_TOS          0x2   /* IP per-packet type of service */
#define   IP_TTL          0x3   /* IP per-packet time to live /
#define   IP_REUSEADDR    0x4   /* allow local address reuse */
#define   IP_DONTROUTE   0x10   /* just use interface addresses */
#define   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   SET_TOS(prec,  tos)   ((0x7 & (prec)) << 5 | (0x1c & (tos)))
```

Abbreviations

CO	Connection-oriented
CL	Connectionless
EM	Event Management
ETSDU	Expedited Transport Service Data Unit
ISO	International Organization for Standardization
OSI	Open System Interconnection
SVID	System V Interface Definition
TC	Transport Connection
TCP	Transmission Control Protocol
TLI	Transport Level Interface
TSAP	Transport Service Access Point
TSDU	Transport Service Data Unit
UDP	User Datagram Protocol
XTI	X/Open Transport Interface
XEM	X/Open Event Management Interface

Appendix H

Minimum OSI Functionality (Preliminary Specification)

H.1 General

The purpose of this specification is to provide a simple API exposing a minimum set of OSI Upper Layers functionality (mOSI).

H.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.

H.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.

H.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 pDISP 11188 — Common Upper Layer Requirements, Part 3: Minimal OSI upper layer facilities — OIW/EWOS working documents). 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.

H.1.4 mOSI API versus XAP

X/Open has developed XAP (ACSE/Presentation API), which offers full access to ACSE and Presentation functionality and is well suited for system programmers/integrators needing to use all of its functionality (including minor and major synchronisation points...).

XAP needs to be used when some of the following pieces of functionality (not available with mOSI) are required:

- use of Functional Units different from Kernel and Full Duplex
- access to AP and AE invocation identifiers, to session connection identifier (which may be useful with some of the resynchronisation/activity management functional units).

In general, XAP will be used for applications targetted at the OSI environment that may need to take advantage of additional OSI facilities in the future. XAP is a flexible, extensible API; extensions to cover OSI Remote Operation Services (ROSE) and OSI Transaction Processing are under development.

H.1.5 Upper Layers Functionality Exposed via mOSI

These are presented as they are exposed via mOSI options and specific parameters.

H.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 datatype, 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 and Application Entity (AE) Qualifier) 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 H.5 on page 283, while its precise structure is implementation dependent.

H.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 H.5 on page 283.

- 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 H.5 on page 283.

- Presentation Context Definition List, 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 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.

H.2 Options

Options are formatted according to the structure t_opthdr as described in Chapter 6 on page 35. An OSI provider compliant to this specification supports all, none or a subset of the options defined in Section H.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 H.1 on page 267.

H.2.1 ACSE/Presentation Connection-oriented Service

The protocol level for all subsequent options is ISO_APCO.

All options are association-related (see Chapter 6 on page 35. 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 H.5 on page 283.

Option Name	Type of Option Value	Legal Option Value	Meaning
AP_CNTX_NAME	Object identifier item (see Section H.5 on page 283)	see text default: see text	Application Context Name
AP_PCDL	Presentation Context Definition list (see Section H.5 on page 283)	see text default: see text	Presentation Context Definition List
AP_PCDRL	Presentation Context Definition Result list (see Section H.5 on page 283)	see text default: none	Presentation Context Definition Result List
AP_MCPC	unsigned long	T_YES / T_NO default:T_NO	multiple choice presentation contexts

Table H-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 H.5 on page 283.

  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 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 H.1 on page 267. The abstract syntax is the default abstract syntax and the transfer syntax is the default transfer syntax, as specified in Section H.5 on page 283.

- Presentation Context Definition Result List

  The codes for the result of negotiation and reason for rejection are defined in Section H.5 on page 283. The responding application, afetr reading this option, may choose to continue or terminate the association.

- Multiple Choice Presentation Contexts

  The default behaviour (AP_MCPC set to T_NO) frees the application from having to make choices for encoding of user-data parameters. In that case, the responder is requested to pick up one of the user-data presentation contexts offered by the initiator; this rule is enforced by the API (Note that the ACSE presentation context is required, but this is handled by the API implementation.

  If a unique user presentation context is too limiting, the application may prefer to perform all encodings (all PDV list, with indication of the PCI used). In this case, on the association responder side, AP_MCPC must be set to T_YES to let the API pass all the user-data presentation contexts offered to the application, responsible for the negotiation (otherwise the API will select a unique one).

  On the association initiator or responder side, when AP_MCPC is set to T_YES, the first user data buffer of each more bit sequence of data buffers starts with a long datatype containing an identifier corresponding to the PCI.

Management Options

No management options are defined.

H.2.2 ACSE/Presentation Connectionless Service

The protocol level for all subsequent options is ISO_APCL.

All options are association-related (see Chapter 6 on page 35). They may be negotiated in all XTI states except T_UNINIT. The structures referenced are specified in Section H.5 on page 283.

Option Name	Type of Option Value	Legal Option Value	Meaning
AP_CNTX_NAME	Object identifier item (see Section H.5 on page 283)	see text default: see text	Application Context Name
AP_PCDL	Presentation Context Definition list (see Section H.5 on page 283)	see text default: see text	Presentation Context Definition List

Table H-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 H.5 on page 283.

  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 List

  In connectionless mode, the transfer syntaxes are not negotiated. Their use are 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 H.1 on page 267). The corresponding abstract and transfer syntaxes are specified in Section H.5 on page 283.

Management Options

No management options are defined.

H.2.3 Transport Service Options

Some of the options defined for XTI ISO Transport Connection-oriented Service or Transport Connectionless 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 190 and Section A.2.2.1 on page 194. 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.

H.3 Functions

t_accept( )

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.

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 and AE Qualifier (see Section H.1 on page 267 and Section H.5 on page 283 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 disconnect reason will be [AC_ABRT_PEER].

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 and Called AE Qualifier.

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-oriented	Connectionless
fd	x	/	/
info->addr	/	x	x
info->options	/	x	x
info->tsdu	/	-1	-1
info->etsdu	/	-2	-2
info->connect	/	x	-2
info->discon	/	x	-2
info->servtype	/	T_COTS_ORD	T_CLTS
info->flags	/	0	0

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 H.1 on page 267 and Section H.5 on page 283.

t_getstate( ) No special considerations for mOSI providers.

t_listen( ) The call->addr structure contains the remote Calling Presentation Address, and optionally the remote Calling AP Title and AE Qualifier.

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-oriented	**Connectionless**
name	x	/	/
oflag		/	/
info->addr	/	x	x
info->options	/	x	x
info->tsdu	/	-1	-1
info->etsdu	/	-2	-2
info->connect	/	x	-2
info->discon	/	x	-2
info->servtype	/	T_COTS_ORD	T_CLTS
info->flags	/	0	0

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 H.2 on page 270.

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.

The call->opt structure may also contain an Application Context Name and/or Presentation Context Definition Result List.

t_rcvdis( ) Possible values for disconnect reason codes are specified in Section H.5 on page 283.

t_rcvrel( ) With this primitive, user data cannot be received on normal release: any user data in the received flow is discarded (see Section H.6 on page 287, XTI Change Request 20-01).

t_rcvudata( ) The unitdata->addr structure specifies the remote Presentation address, and optionally the remote AP Title and AE Qualifier. 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. 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 Section H.6 on page 287, XTI Change Request 20-01).
t_sndudata( )	The unitdata->addr structure specifies the remote Presentation address, and optionally the remote AP Title and AE Qualifier. 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.

H.4 Implementors´ Notes

H.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.

H.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 following parameters must be explicitly set by the application:
  - Remote (called) Presentation Address (in *t_connect*( ), sndcall->addr).
  - If, in *t_accept*( ), a new accepting endpoint is specified (resfd != fd), a Presentation Address must be bound to the new accepting endpoint (the Responding Presentation Address). If the same endpoint is used, the Responding Presentation Address is equal to the Local (Called) Presentation Address.
- 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 H.2 on page 270):
  - Application Context Name (opt parameter)
  - Presentation Contexts (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*).

- 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
  - — AP invocation identifiers (called, calling, and responding)
  - — AE invocation identifiers (called, calling, and responding)
  - — Session connection identifiers.

H.4.3 Mapping XTI Functions to ACSE/Presentation Services

In the following tables, for a given primitive, the presence of each parameter in the protocol flow is described in the OSI column by M or O, as specified in Annex A of *Common Upper Layer Requirements, Part 3: Minimal OSI upper layer facilities* - *OIW/EWOS working document*. Connectionless protocols are not yet included in CURL part 3.

For items sent, the status column is from the *Sender Status for Category II specification* column in Annex A of CURL - part 3. The Receiver Status is always set to *M*, as parameters which are optional on the sending side must be acceptable (that is, not generate aborts) on the receiving side, even if they are subsequently to be ignored.

M Mandatory: Support for the feature is mandatory — as sender, as receiver or as both sender and receiver.

O Optional: Support for the item is the option of the referencing specification — as sender, as receiver or as both sender and receiver.

H.4.3.1 Connection-oriented 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 H-3 Association Establishment

XTI call	Parameter	Service	Parameter	OSI
t_connect		A-ASSOCIATE req		
	sndcall->addr		Called Presentation Address	M
	sndcall->addr (1)		Called AP Title	O
	sndcall->addr (1)		Called AE Qualifier	O
	sndcall->opt (2)		Application Context Name	M
	sndcall->opt (3)		P-context Definition List	M
	sndcall->udata		User Information	O
{t_bind}	req | ret->addr		Calling Presentation Address	M
{t_bind}	req | ret->addr		Calling AP Title	O
{t_bind}	req | ret->addr		Calling AE Qualifier	O
t_listen		A-ASSOCIATE ind		
	call->addr		Calling Presentation Address	M
	call->addr (1)		Calling AP Title	M
	call->addr (1)		Calling AE Qualifier	M
	call->opt		Application Context Name	M
	call->opt (4)		P-context Definition List	M
	call->udata		User Information	M
{t_bind}	req | ret->addr		Called Presentation Address	M
{t_bind}	req | ret->addr (1)		Called AP Title	M
{t_bind}	req | ret->addr (1)		Called AE Qualifier	M
t_accept		A-ASSOCIATE rsp+		
	call->addr		not used: Calling Presentation Address	O
	call->opt		Application Context Name	M
	call->opt		P-context Definition Result List	M
	call->udata		User Information	O
{internal}	::="accepted"		Result	M
{t_bind}	req | ret->addr		Responding Presentation Address	M
{t_bind}	req | ret->addr (1)		Responding AP Title	O
{t_bind}	req | ret->addr (1)		Responding AE Qualifier	M
not sent		A-ASSOCIATE rsp-		
t_connect	(synchronous mode)	A-ASSOCIATE cnf+		
	rcvcall->addr		Responding Presentation Address	M
	rcvcall->addr		Responding AP Title	M
	rcvcall->addr		Responding AE Qualifier	M
	rcvcall->opt		Application Context Name	M
	rcvcall->opt		P-context Definition Result List	M
	rcvcall->udata		User Information	M
{internal}	::="accepted"		Result	M
{internal}	::="ACSE service-user"		Result Source	M
t_rcvconnect	(asynchronous mode)	A-ASSOCIATE cnf+		
	call->addr		Responding Presentation Address	M
	call->addr		Responding AP Title	M
	call->addr		Responding AE Qualifier	M
	call->opt		Application Context Name	M
	call->opt		P-context Definition Result List	M
	call->udata		User Information	M
{discarded}	::="accepted"		Result	M
{discarded}	::="ACSE service-user"		Result Source-diagnostic	M
t_rcvdis		A-ASSOCIATE cnf-		

XTI call	Parameter	Service	Parameter	OSI
	discon->udata		User Information	M
	discon->reason (5)		Result	M
{internal}	ACSE serv-user | pres serv-prov		Result Source-diagnostic	M
	{discarded}		Application Context Name	M
	{discarded}		P-context Definition Result List	M

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	OSI
t_snd		P-DATA req		
	buf		User Data	M
t_rcv		P-DATA ind		
	buf		User Data	M

Table H-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_sndrel2*( ) and *t_rcvrel2*( ), see Section H.6 on page 287). 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	OSI
t_sndrel2		A-RELEASE req		
	reldata->reason		Reason	M
	reldata->udata		User Information	O
t_rcvrel2		A-RELEASE ind		
	reldata->reason		Reason	M
	reldata->udata		User Information	M
t_sndrel2		A-RELEASE rsp		
	reldata->reason		Reason	M
	reldata->udata		User Information	O
t_rcvrel2		A-RELEASE cnf		
	reldata->reason		Reason	M
	reldata->udata		User Information	M
t_snddis		A-ABORT req		
	n/s		Diagnostic	M
	call->udata		User Information	O
t_rcvdis		A-ABORT ind		
	discon->reason		Diagnostic	M
	discon->udata		User Information	M
t_rcvdis		A-P-ABORT ind		
	discon->reason		Diagnostic	M

Table H-5 Association Release

H.4.3.2 Connectionless Services

XTI call	Parameter	Service	Parameter	OSI
t_sndudata		A-UNIT-DATA source		
	unitdata->addr		Called Presentation Address	M
	unitdata->addr		Called AP Title	O
	unitdata->addr		Called AE Qualifier	O
	unitdata->opt (1)		Application Context Name	M
	unitdata->opt (2)		P-context Definition List	O (4)
	unitdata->udata		User Information	M
{t_bind}	req | ret->addr		Calling Presentation Address	M
{t_bind}	req | ret->addr		Calling AP Title	O
{t_bind}	req | ret->addr		Calling AE Qualifier	O
t_rcvudata		A-UNIT-DATA sink		
	unitdata->addr		Calling Presentation Address	M
	unitdata->addr		Calling AP Title	M
	unitdata->addr		Calling AE Qualifier	M
	unitdata->opt		Application Context Name	M
	unitdata->opt (3)		P-context Definition List	M (4)
	unitdata->udata		User Information	M
{t_bind}	req | ret->addr		Called Presentation Address	M
{t_bind}	req | ret->addr		Called AP Title	M
{t_bind}	req | ret->addr		Called AE Qualifier	M

Table H-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

(4) ISO 8822 AM1 (connectionless Presentation service) defines this parameter as *user option*; CURL part 3 does not currently cover connection-less services.

H.5 Complements to <xti.h>

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 {
    unsigned int    osi_apt_len;
    unsigned int    osi_aeq_len;
    unsigned int    osi_paddr_len;
    unsigned char   osi_addr[ MAX_ADDR ];
}
```

Where:

the *apt* address starts at osi_addr[0]
the *eaq* address starts at osi_addr[T_ALIGN(osi_apt_len)]
the *paddr* is at osi_addr[T_ALIGN(osi_apt_len) + T_ALIGN(osi_aeq_len)]
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 ISO_APCO        0x0200
#define ISO_APCL        0x0300
#define AP_CNTX_NAME    0x1
#define AP_PCDL         0x2
#define AP_PCDRL        0x3
#define AP_MCPC         0x4
```

Object Identifier Representation within Options

The presentation context definition list and application context both utilise object identifiers. An object identifier is held as a variable length item of the following form:

```
| len | object_value.... | // |
 -----------------------------
 ulong                     ^
                           |
                       alignment
                       characters
```

The application is responsible for encoding/decoding the Object id 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 List Option

The presentation context definition list option is used to propose one or more presentation contexts, giving their abstract syntax and allowable transfer syntaxes.

The presentation context definition list option is a variable size option consisting of a long giving the number of presentation contexts followed that number of presentation context definition elements.

Each presentation context definition element consists of a presentation context item header defined as:

```
struct t_ap_pcd_hdr {
    long pci;
    long t_sytx_size;
}
```

followed by an object identifier item for the abstract syntax and *t_sytx_size* number of object identifier items, one for each of the proposed transfer syntaxes.

Presentation Context Definition Result List Option

A presentation definition context result list option gives the result of negotiation, and consists of a long giving the number of presentation contexts followed by that number of presentation context definition result elements, each defined as:

```
struct t_ap_pcdr {
    long res;         /* result of negotiation */
    long prov_rsn;    /* reason for rejection  */
}

/*
* codes for res and prov_rsn
*/

#define PCDRL_ACCPT      0x0
                        /*pres. context accepted                   */
#define PCDRL_USER_REJ   0x1
                        /*pres. context rejected by peer application */
#define PCDRL_PREJ_RSN_NSPEC  0x0100
                        /*prov. reject: no reason specified         */
#define PCDRL_PREJ_A_SYTX_NSUP  0x0101
                        /*prov. reject: abstract syntax not supported*/
#define PCDRL_PREJ_T_SYTX_NSUP  0x0102
                        /*prov. reject:transfer syntax not supported */
#define PCDRL_PREJ_LMT_DCS_EXCEED  0x0103
                        /*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, 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 CURL 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 CURL 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 AC_U_AARE__NONE   0x0001     /*connection rejected by      */
                                     /*peer user: no reason given  */
#define AC_U_AARE_ACN     0x0002     /*connection rejected:        */
                                     /*application context name    */
                                     /*not supported               */
#define AC_U_AARE_APT     0x0003     /*connection rejected:        */
                                     /*AP title not recognised     */
#define AC_U_AARE_AEQ     0x0005     /*connection rejected:        */
                                     /*AE qualifier not recognised */
#define AC_U_AARE_PEER_AUTH 0x000e   /*connection rejected:        */
                                     /*authentication required     */
#define AC_P_ABRT_NSPEC   0x0011     /*aborted by peer provider:   */
                                     /*no reason given             */
#define 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.

H.6 XTI mOSI CR

Several references are made in this Appendix to XTI CR 20-01. This change request proposes extending the functionality of the X/Open Transport interface. It is intended that this change proposal will be decided along with other proposals to extend XTI functionality, in the near future.

The content of XTI CR 20-01 is presented in this section, for convenience. If accepted, it will be removed from this Appendix when the extended functionality it proposes is incorporated with other XTI functions.

```
Document:       X/Open Transport Interface (XTI), CAE Specification, Version 2

Change Number: 20-01

Title:          Orderly release with user data

Qualifier:      Major Technical

Rationale:      XTI permits to send and receive user data with the
                abortive release primitives (t_snddis, t_rcvdis) but not
                with the orderly release primitives (t_sndrel, t_rcvrel).

                This is consistent with TCP specifications.

                For ISO ACSE, providing an orderly release mechanism, user
                data is a parameter of the release service.  OSI
                applications that use A-RELEASE user data are FTAM and VT
                (Virtual Terminal); for ROSE applications, the argument
                of UNBIND is mapped to A-RELEASE user data.

                When mapping XTI primitives to ACSE/Presentation
                (XTI-mOSI Appendix), disconnect user data may thus be
                received from peer applications.  Three alternatives are
                possible:

                1. Discard user data - detrimental to those applications
                   that want to receive user data from the peer, but
                   still possible for the others.
                2. User data delivered (via t_rcv); this would happen
                   just before the T_ORDREL event, with introduction of a
                   new flag T_ORDREL_DATA.
                3. Addition of a new primitive, t_rcvrel2, with user data
                   parameter - this method is more straightforward than
                   alternative 2 (both for the application and the
                   library implementation), and additive (does not break
                   existing applications).
                See part 1 of the proposed change.

                If user data can be received with t_rcvrel2, it makes sense
                to propose a similar handling for the emission of user data,
                thus a new primitive t_sndrel2, with user data parameter, is
                proposed below (if this new primitive is not present, the
                only alternative is that user data is not supported by the
                provider). See part 2 of the proposed change.

                Support of these new primitives needs to be indicated,
                see part 3 of the change.
```

Change:

```
Part 1: Add a new manual page after t_rcvrel():
-----------------------------------------------

Name:
  t_rcvrel2 - receipt of an orderly release indication or
  confirmation containing user data.

SYNOPSIS
  #include <xti.h>

  int t_rcvrel2(fd, discon)
  int fd;
  struct t_discon *discon;

DESCRIPTION

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

  This function is used to acknowledge receipt of an
  orderly release indication or confirmation 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 because such an attempt
  will block forever.  However, the user may continue to
  send data over the connection if t_sndrel() or t_sndrel2()
  has not been called by the user.

  The field reason specifies the reason for the
  disconnect through a protocol-dependent reason code
  and udata identifies any user data that was sent with the
  disconnect; 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 disconnect will be discarded.

  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 does not discard received disconnect user data.
```

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.

[TNOREL] No orderly release indication currently exists on the specified transport endpoint.

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

[TNOTSUPPORT] Orderly release is not supported by the underlying transport provider.

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

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

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

[TBUFOVFLW] The number of bytes allocated for incoming data (maxlen) is greater than 0 but not sufficient to store the data, and the disconnect 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.

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_sndrel2(), t_rcvrel(), t_sndrel().

Part 2: Add a new manual page after t_sndrel():

Name:
t_sndrel2 - initiate/respond to an orderly release with user data

SYNOPSIS

```
#include <xti.h>

int t_sndrel2(fd, discon)
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 or to respond to an orderly release indication 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_sndrel2(), 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 disconnect through a protocol-dependent reason code and udata identifies any user data that is sent with the disconnect; 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 dicson filed 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.

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 will accept to send disconnect 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:

[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 function was issued in the wrong sequence on the transport endpoint referenced by fd.
[TSYSERR]	A system error has occurred during execution of this function.
[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI(t_errno).
[TBADDATA]	The amount of user data specified was not within the bounds allowed by the transport provider or the provider did not return T_ORDRELDATA in the t_open flags.

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_rcvrel2(), t_rcvrel(), t_sndrel().

```
Part 3: indication by provider of support of the new
        primitives
-------------------------------------------------------------

- Section 4.3, XTI features:
Change:
"The orderly release mechanism (using t_sndrel() and
t_rcvrel()) is supported only for T_COTS_ORD type providers."

into:
"The orderly release mechanism (using t_sndrel(), t_sndrel2(),
t_rcvrel() and t_rcvrel2()) is supported only for T_COTS_ORD
type providers."


Other sections with editorial changes resulting from this CR:
-------------------------------------------------------------
Mention t_rcvrel2(), t_sndrel2() in addition to existing
functions in:
- Table 3-1,
- Section 4.1.4, Overview of Connection Release,
- Section 4.3.1, XTI Functions versus Protocols,
- Table 5-2,
- Section 5.6, Events and TLOOK error indication.
```

Appendix I

SNA Transport Provider

I.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 will be published in 1993. Copies are available[9] on request.

---

9. Until the full-duplex verbs are published in the public domain, copies of the relevant specification *CPI-C Full Duplex Conversations and Expedited Data, Nov 30 1992* may be requested from IBM Corporation, via X/Open.

I.2 SNA Transport Protocol Information

This section describes the protocol-specific information that is relevant for Systems Network Architecture (SNA) transport providers.

I.2.1 General

1. Protocol address

   For information about SNA addresses, see Section I.2.2 on page 295.

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 disconnect reasons returned on *t_rcvdis*( ) include [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. Orderly release

   The orderly release functions, *t_sndrel*( ) and *t_rcvrel*( ), can be used for the orderly release facility of SNA, just as they are for TCP.

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 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.

I.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 SNA_MAX_NETID_LEN   8
#define SNA_MAX_LU_LEN      8
#define SNA_MAX_TPN_LEN

struct sna_addr
   {
     u_char  sna_netid    (SNA_MAX_NETID_LEN),
     u_char  sna_lu       (SNA_MAX_LU_LEN),
     u_short sna_tpn_len,      /* less than or equal to 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.

I.2.3 Options

Options are formatted according to the structure **t_opthdr** as described in Chapter 6. A transport provider compliant to this specification supports none, all, or any subset of the options defined in Section I.2.3.1.

I.2.3.1 Connection-Mode Service Options

The protocol level of all subsequent options is SNA.

All options are association-related. 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 I-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
SNA_MODE	char	SNA_BATCH SNA_BATCHSC SNA_INTER SNA_INTERSC SNA_DEFAULT any user-defined SNA mode value	SNA mode, which controls the underlying class of service selected for the connection. The SNA mode is specified only by the active side of the connection. If not specified, the default mode is SNA_DEFAULT. The default mode characteristics may vary from system to system.
SNA_ALWAYS_PUSH	unsigned long	T_YES / T_NO	If T_YES, every *t_snd*( ) operation will cause the message to be sent immediately. 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. Default value is T_NO. This option is primarily for programs migrated to SNA from other protocol stacks that always send data immediately. It allows them to request behaviour similar to that on the original provider. However, setting SNA_ALWAYS_PUSH to T_YES may affect its performance.

Table I-1 SNA Options

I.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	/	-1
info-->etsdu	/	-2 / 86 [2]
info-->connect	/	-2
info-->discon	/	-2
info-->servtype	/	T_COTS_ORD
info-->flags	/	T_SNDZERO

Table I-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	/	-1
info-->etsdu	/	-2 / 86 [2]
info-->connect	/	-2
info-->discon	/	-2
info-->servtype	/	T_COTS_ORD
info-->flags	/	T_SNDZERO

Table I-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 disconnect reason codes are valid for any implementation of an SNA transport provider under XTI:

```
#define SNA_CONNECTION_SETUP_FAILURE.
#define SNA_USER_DISCONNECT
#define SNA_SYSTEM_DISCONNECT
#define SNA_TIMEOUT
#define SNA_CONNECTION_OUTAGE
```

These definitions should be included in **<xti.h>**.

t_snd( ) Unless the SNA_ALWAYS_PUSH option is set to T_YES or the T_PUSH flag on the *t_snd*( ) function is set, the SNA transport provider may collect data in a send buffer until it accumulates a sufficient amount for transmission. The amount of data that is accumulated can vary from one connection to another.

In order to take advantage of the performance improvement that SNA buffering offers, the XTI user must set the SNA_ALWAYS_PUSH option to T_NO (Default is T_YES). If this option is not set to T_NO, a push will be done for every *t_snd*( ) and the T_PUSH flag will have no effect.

t_snddis( ) Since no user data is sent during a disconnect 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 SNA_INTERSC or SNA_BATCHSC modes.

I.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.

I.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 I-4 gives an explanation for each FDX verb that is used in these mappings.

Table I-4 FDX LU 6.2 Verb Definitions

FDX Verb	Description
ALLOCATE	Allocates a conversation between the local transaction program and a remote (partner) transaction program.
DEALLOCATE	DEALLOCATE with TYPE(FLUSH) closes the local program's send queue. Both the local and remote program must close their send queues independently. DEALLOCATE with TYPE(ABEND_PROG) is an abrupt termination that will close both sides of the conversation simultaneously.
FLUSH	Flushes the local LU's send buffer.
GET_ATTRIBUTES	Returns information pertaining to the specified conversation.
GET_TP_PROPERTIES	Returns information pertaining to the transaction program issuing the verb.
RECEIVE_ALLOCATE	Receives a new conversation with a partner transaction program that issued ALLOCATE.
RECEIVE_AND_WAIT	Waits for data to arrive on the specified conversation and then receives the data. If data is already available, the program receives it without waiting.
RECEIVE_EXPEDITED_DATA	Receives data sent by the remote transaction program in an expedited manner, via the SEND_EXPEDITED_DATA verb.
SEND_DATA	Sends data to the remote transaction program.
SEND_EXPEDITED_DATA	Sends data to the remote transaction program in an expedited manner. This means that it may arrive at the remote transaction program before data sent earlier via a send queue verb; for example, SEND_DATA.
WAIT_FOR_COMPLETION	Waits for posting to occur on one or more non-blocking operations represented in the specified wait objects. Posting of a non-blocking operation occurs when the LU has completed the associated non-blocking verb and filled all the return values.

I.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 I-1 is used for blocking XTI calls; the second Figure I-2 is used for non-blocking calls.

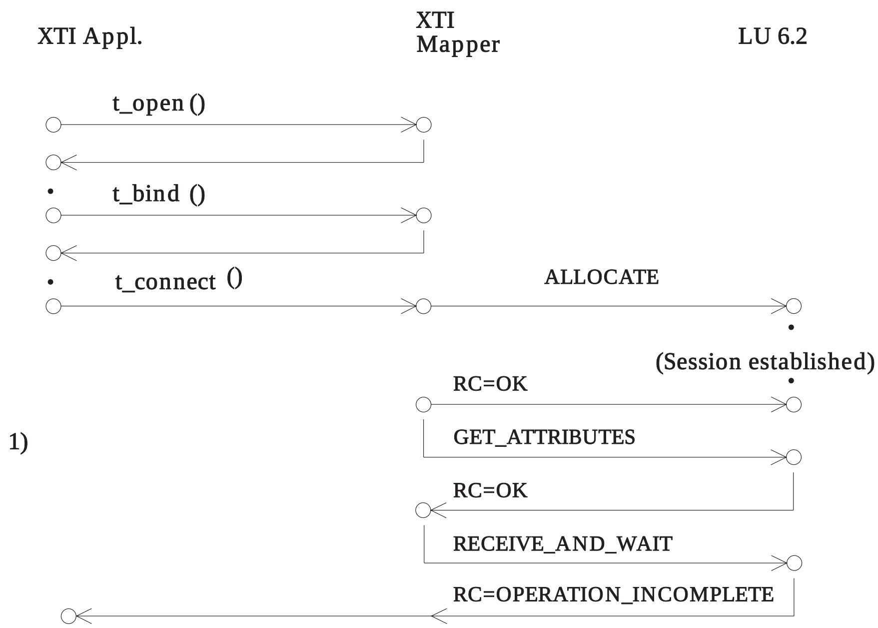


Figure I-1 Active Connection Establishment, Blocking Version (1 of 2)

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*.

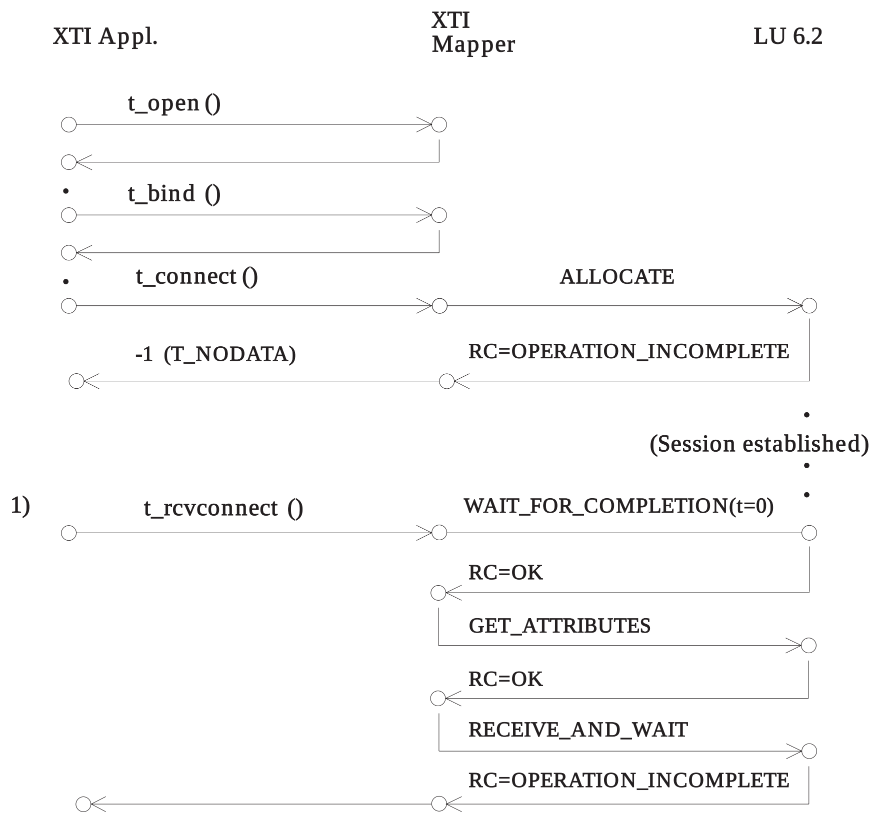


Figure I-2 Active Connection Establishment, Non-blocking Version (2 of 2)

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 I-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 I-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 I-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.

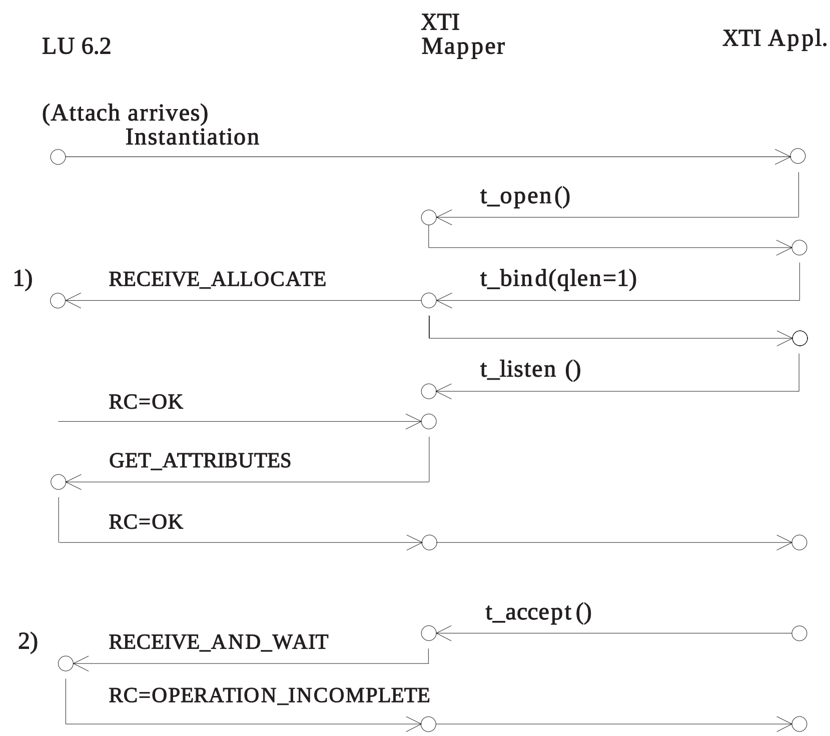


Figure I-3 Passive Connection Establishment, Instantiation Version (1 of 3)

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.

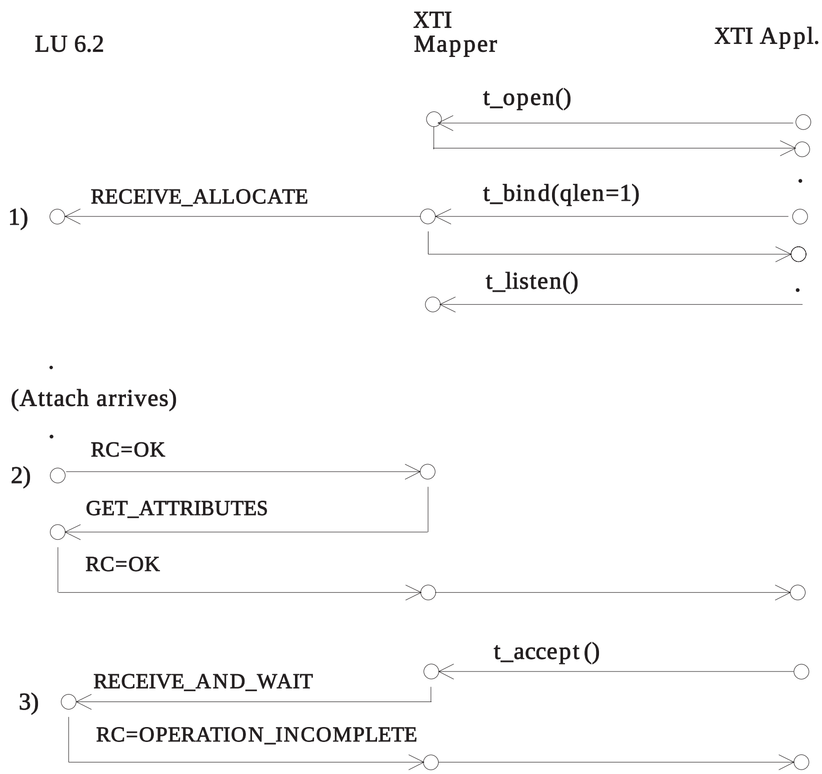


Figure I-4 Passive Connection Establishment, Blocking Version (2 of 3)

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.

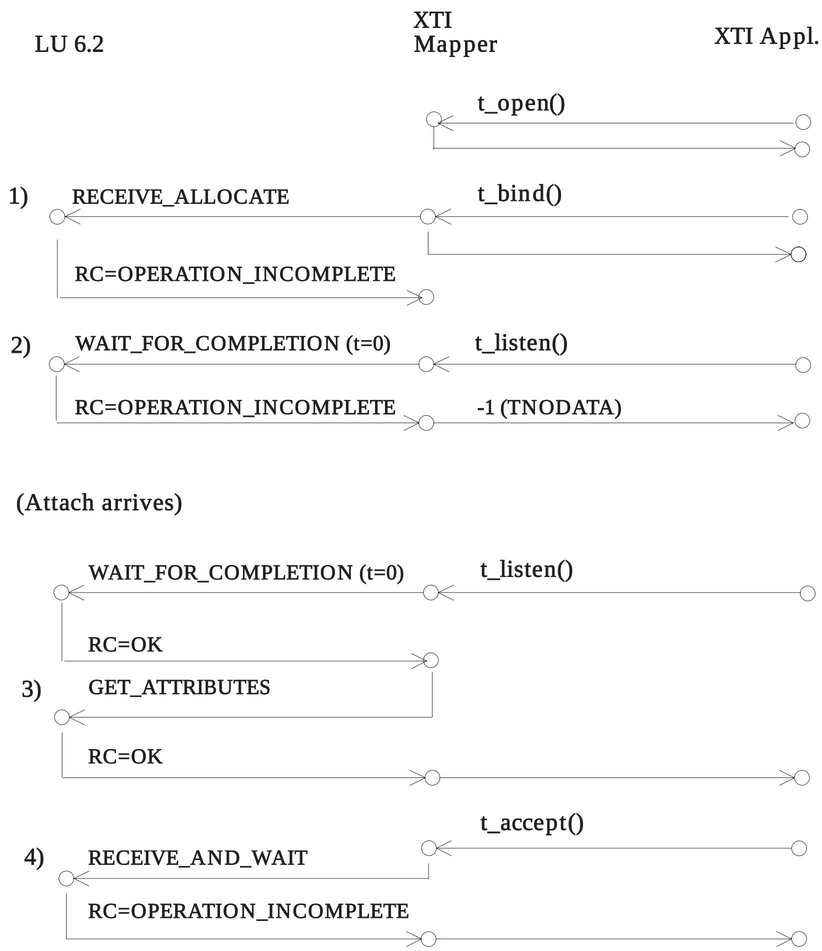


Figure I-5 Passive Connection Establishment, Non-blocking Version (3 of 3)

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 connect request has been received. The *t_listen* will cause a WAIT_FOR_COMPLETION, with time=0, to be issued. The WAIT_FOR_COMPLETION will check on the wait object from the previous non-blocking RECEIVE_ALLOCATE. In this example, when the first *t_listen* is issued, the RECEIVE_ALLOCATE is still outstanding; but the RECEIVE_ALLOCATE has completed before the second *t_listen* is issued.

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 I-6, shows the mapping for the blocking XTI *t_snd*( ) call. The diagram after this, Figure I-7, shows the non-blocking mapping of the XTI *t_snd*( ) call.

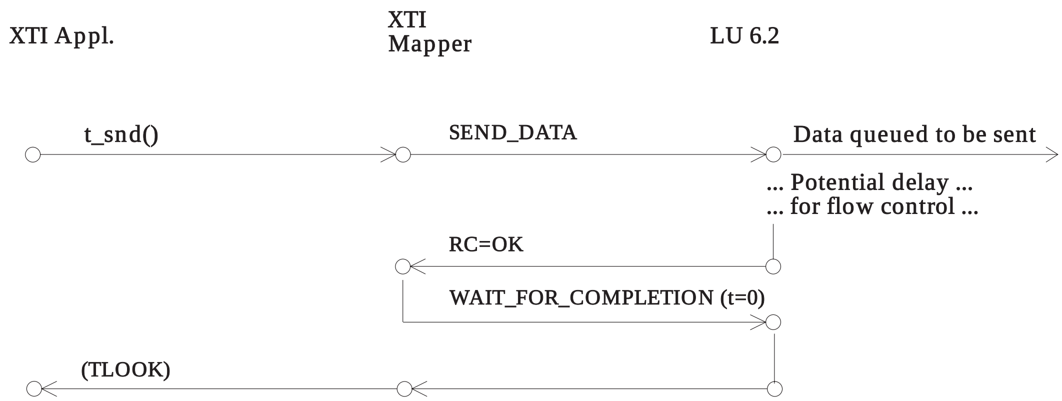


Figure I-6 XTI Function to LU 6.2 Verb Mapping: Blocking t_snd

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.

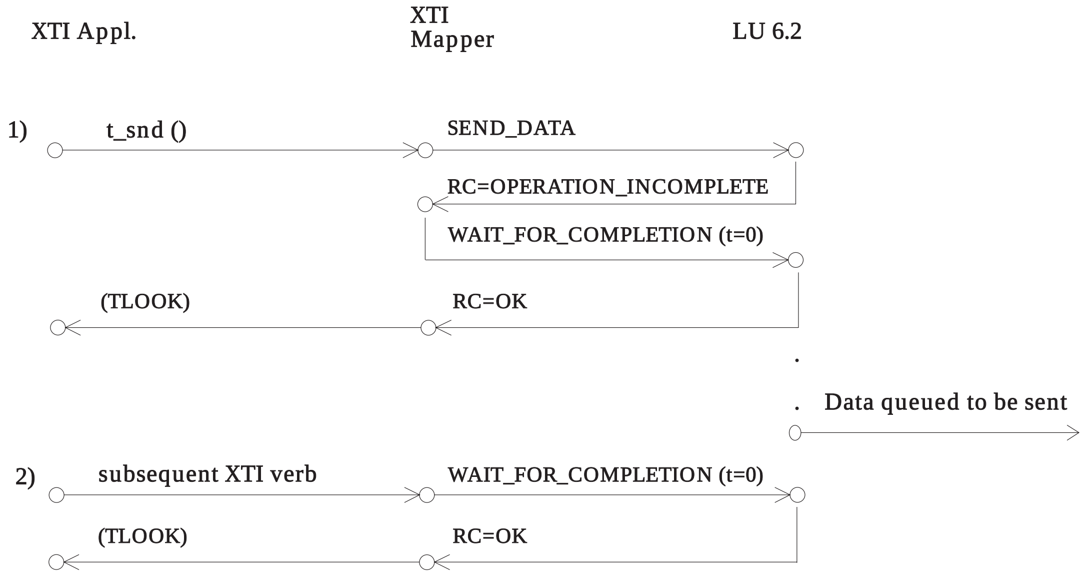


Figure I-7 XTI Function to LU 6.2 Verb Mapping: Non-blocking t_snd

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 I-8, shows the mapping for blocking XTI receive call, and the diagram after this, Figure I-9, shows the mapping for non-blocking XTI receive call.

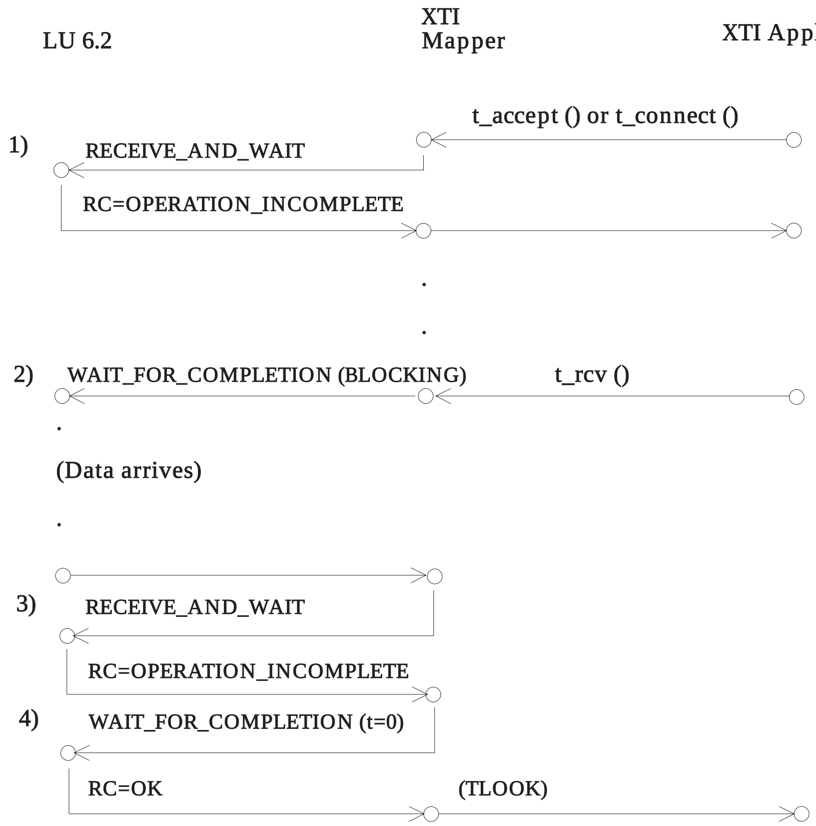


Figure I-8 XTI Function to LU 6.2 Verb Mapping: Blocking t_rcv

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.

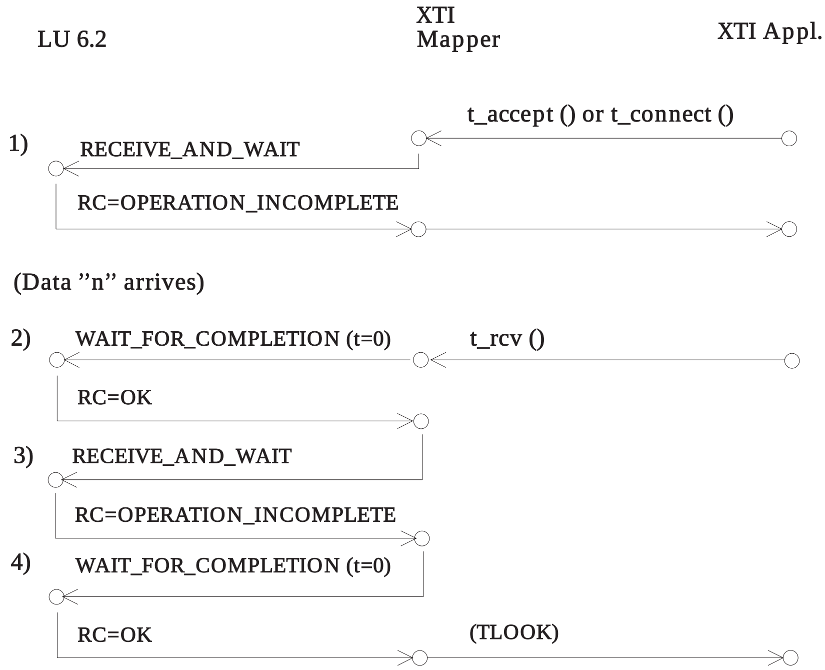


Figure I-9 Mapping from XTI Calls to LU 6.2 Verbs (Passive side)

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.

I.3.3 Full Duplex Mapping

The following table shows the mapping from XTI function calls to full duplex LU 6.2 verbs.

Table I-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. Refer to Table I-7 on page 314.
t_alloc( )	Local	
t_bind( )	If qlen>0: RECEIVE_ALLOCATE for each connection request that can be queued. Optionally: DEFINE_TP 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 I-3 on page 305.)	Refer to Table I-8 on page 315.
t_close( )	If connection still up issue DEALLOCATE TYPE(ABEND)	May be a delay if XTI_LINGER option activated with non-zero linger value. Refer to Table I-9 on page 315.
t_connect( )	ALLOCATE RETURN_CONTROL GET_ATTRIBUTES RECEIVE_AND_WAIT	Refer to Table I-10 on page 316.
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 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. Refer to Table I-11 on page 318.
t_getstate( )	Local	
t_listen( )	WAIT_FOR_COMPLETION	Refer to Table I-12 on page 319.
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 Refer to Table I-13 on page 319.

XTI Function	**SNA FDX LU6.2 verb**	**Comments**
t_rcv( )	WAIT_FOR_COMPLETION RECEIVE_AND_WAIT [RECEIVE_EXPEDITED_DATA] WAIT_FOR_COMPLETION	Refer to Table I-14 on page 320.
t_rcvconnect( )	WAIT_FOR_COMPLETION GET_ATTRIBUTES RECEIVE_AND_WAIT	Refer to Figure I-2 on page 304.
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) [FLUSH] [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. A zero-length TSDU causes the following LL to be sent: hex 0002. This can be used to turn off the LL continuation set on the previous send. Refer to Table I-16 on page 322.
t_snddis( )	DEALLOCATE TYPE(ABEND_PROG)	Takes down both directions of the connection Refer to Table I-17 on page 323.
t_sndrel( )	DEALLOCATE TYPE(FLUSH)	Takes down send direction of conversation only. Refer to Table I-19 on page 323.
t_sndudata( )	SEND_DATA on datagram server conversation	Refer to Table I-19 on page 323.
t_strerror( )	local	
t_sync( )	Local	
t_unbind( )	Local	

I.3.3.1 Parameter Mappings

Table I-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 I-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 I-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) TP_NAME(variable)
req-->qlen	Used Locally	>0, RECEIVE_ALLOCATE issued for each connect request that can be queued.
ret-->addr.maxlen	Used Locally	
	Constant	RETURN_CONTROL (WHEN_ALLOCATE_RECEIVED)
	Constant	SCOPE(ALL)
	Created Locally	WAIT_OBJECT(BLOCKING)
Output		
ret-->addr.len	Created Locally	
ret-->addr.buf	---->	LOCAL_LU_NAME(variable) TP_AL_LU_NAME(variable)
ret-->addr.qlen	Created Locally	
	Used Locally	RESOURCE(variable)
t_errno	---->	RETURN_CODE(variable)

Table I-9 *t_close* <--> FDX Verbs and Parameters

XTI Function and Parameters	*<--Relation-->*	**FDX Verb & Parameter**
t_close **If connection is still in T_DATAXFER state**		**DEALLOCATE**
Input		
fd	---->	RESOURCE(variable)
	Constant	TYPE(ABEND_PROG)
Output		
t_errno	<----	RETURN_CODE(variable)

Table I-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) If platform does not support this tower (Tower 205), use (WHEN_SESSION_ALLOCATED).
	Created Locally	WAIT_OBJECT(BLOCKING) if blocking WAIT_OBJECT(VALUE(variable)) if non-blocking
Output		
	Used Locally	RESOURCE(variable)
t_errno	<----	
t_connect		**GET_ATTRIBUTES**
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
Output		
rcvcall-->addr.len	<----	PARTNER_FULLY_QUALIFIED_ LU_NAME(variable)
rcvcall-->addr.buf	<-----	PARTNER_FULLY_QUALIFIED_ LU_NAME(variable)
rcvcall-->opt.len	<----	MODE_NAME(variable)
rcvcall-->opt.buf	<----	MODE_NAME(variable)
t_errno	<----	RETURN_CODE(variable)
t_connect		**RECEIVE_AND_WAIT**
Input		

XTI Function and Parameters	<--*Relation*-->	**FDX Verb & Parameter**
fd	Used Locally	
	Created Locally	RESOURCE(variable)
Output		
	Created Locally	LENGTH(variable)
	Created Locally	FILL(addr of local bufr)
	Constant	WAIT_OBJECT(BLOCKING)

Table I-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) OWN_TP_NAME(variable)
t_errno	<----	RETURN_CODE(variable)

Table I-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 I-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 I-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. 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 • T_MORE • T_EXPEDITED=NO / YES	Created Locally	WHAT_RECEIVED(variable) 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 I-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 I-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 • T_EXPEDITED=NO • T_MORE • T_FLUSH	---->	LL continuation bit
	Created Locally	WAIT_OBJECT(BLOCKING) if blocking WAIT_OBJECT(VALUE(variable)) if non-blocking
Output		
(TLOOK)	<----	EXPEDITED_DATA_RECEIVED
t_errno	<----	RETURN_CODE
t_snd()		**SEND_EXPEDITED_DATA**
Input		
fd	Used Locally	
	Created Locally	RESOURCE(variable)
buf	---->	DATA(variable)
nbytes	---->	LENGTH(variable)
flags • T_EXPEDITED=YES • T_MORE • T_FLUSH	---->	LL continuation bit 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 I-17 *t_snddis* (Existing Connection) <--> FDX Verbs and Parameters

XTI Function and Parameters	<--*Relation*-->	**FDX Verb & Parameter**
t_snddis() **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 I-18 *t_snddis* (Incoming Connect Req.) <--> FDX Verbs and Parameters

XTI Function and Parameters	<--*Relation*-->	**FDX Verb & Parameter**
t_snddis() **To reject incoming connect request**		DEALLOCATE
Input		
call-->sequence	Used Locally	
Output		
t_errno	<----	RETURN_CODE(variable)

Table I-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)

I.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.

I.3.5 Return Code to Event Mapping

The following table Table I-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 I-20 Mapping of XTI Events to SNA Events

XTI Event	Full Duplex SNA Event
T_CONNECT	ALLOCATE completes with RETURN_CODE=OK
T_DATA	RECEIVE_AND_WAIT completes with OK return code and WHAT_RECEIVED = • DATA_COMPLETE • DATA_INCOMPLETE
T_DISCONNECT	One of the following has occurred: SEND_DATA issued with RETURN_CODE = ERROR_INDICATION with subcode from list below: • ALLOCATION_ERROR • DEALLOCATE_ABEND_PROG • DEALLOCATE_ABEND_SVC • DEALLOCATE_ABEND_TIMER • RESOURCE_FAILURE_NO_RETRY • RESOURCE_FAILURE_RETRY Any other verb issued with RETURN_CODE of • ALLOCATION_ERROR • DEALLOCATE_ABEND_PROG • DEALLOCATE_ABEND_SVC • DEALLOCATE_ABEND_TIMER • RESOURCE_FAILURE_NO_RETRY • RESOURCE_FAILURE_RETRY
T_EXDATA	RECEIVE_EXPEDITED_DATA completes with OK return code
T_GODATA	Flow control restrictions on normal data flow that lead to a [TFLOW] error have been lifted. Normal data may be sent again.
T_GOEXDATA	Flow control restrictions on expedited data flow that lead to a [TFLOW] error have been lifted. Expedited data may be sent again.
T_LISTEN	When the partner program is instantiated when the connection request arrives (the typical LU 6.2 model), this event is posted as soon as the program issues the *t_listen*( ) function call. When the partner program already exists, this event is posted when the connection request arrives and is matched with the program.
T_ORDREL	Set when a RECEIVE_* verb completes with RETURN_CODE = DEALLOCATE_NORMAL.

I.4 XTI SNA CR

Reference is made in this Appendix (see item 7 in Section I.2.1 on page 294, and the description for *snd*( ) in Section I.2.4 on page 298) to a T_PUSH flag on the *t_snd*( ) function. The change request below proposes extending the functionality of the X/Open Transport Interface, to add this T_PUSH flag. It is intended that this change proposal will be decided along with other proposals to extend XTI functionality, in the near future.

For convenience, a copy of this CR, numbered 20-02 for the purposes of this publication, is included in this Appendix. If accepted, this CR will be removed from this Appendix when the extended functionality it proposes is incorporated into the XTI *snd*( ) function description in *t_snd*( ) on page 93.

```
Document:      X/Open Transport Interface (XTI), CAE Specification, Version 2

Change Number: 20-02

Title:         T_PUSH flag needed on t_snd().

Qualifier:     Major Technical

Rationale:     In the XTI Appendix I, SNA Transport Provider, reference
               is made to a T_PUSH flag on the XTI t_snd() function.
               This CR proposes change to add the T_PUSH flag to the
               t_snd() function.

Change:        Add the following text to the description of the flags
               on the t_snd() definition.
               This will be the third flag on the t_snd() function
               description in the XTI specification.

                T_PUSH   If set in flags, the transport provider
                         will flush all data that is currently
                         in its send buffers. If not set in flags,
                         the transport provider is free to collect
                         data in a send buffer until it accumulates
                         a sufficient amount for transmission.
```

Appendix J

The Internet Protocols

The Internet Protocol (IP) family is a collection of protocols designed for use in the Internet and using the Internet address format. The Internet family provides protocol support for the **SOCK_STREAM** and **SOCK_DGRAM** socket types.

Internet addresses are 4-byte quantities, stored in network byte order (on ''little-endian'' machines, these are word and byte reversed). The **<netinet/in.h>** header defines this address as a discriminated union.

The address INADDR_ANY can be given in a *bind*( ) call on a socket that uses the TCP or UDP protocol. For TCP, this lets the socket accept connections targeted at any of the host's IP addresses. For UDP, it lets the socket accept packets addressed to any of the host's IP addresses. The address INADDR_BROADCAST is the IP broadcast address. If INADDR_BROADCAST is used in a *sendto*( ) call on a UDP socket, and the host has one or more network interfaces that support the broadcast feature, a broadcast packet will be sent via one or more of those interfaces.

The Internet protocol family includes the following protocols:

- the IP transport protocol
- the Internet Control Message Protocol (ICMP)
- the Transmission Control Protocol (TCP)
- the User Datagram Protocol (UDP).

TCP is used to support the SOCK_STREAM abstraction while UDP is used to support the SOCK_DGRAM abstraction.

The 32-bit Internet address contains both network and host parts. It is frequency-encoded; the most-significant bit is clear in Class A addresses, in which the high-order 8 bits are the network number. Class B addresses use the high-order 16 bits as the network field, and Class C addresses have a 24-bit network part. Sites with a cluster of local networks and a connection to the Internet may choose to use a single network number for cluster; this is done by using subnet addressing. The local (host) portion of the address is further subdivided into subnet and host parts. Within a subnet, each subnet appears to be an individual network; externally, the entire cluster appears to be a single, uniform network requiring only a single routing entry.

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.

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.

expedited data
Data that are considered urgent. The specific semantics of expedited data are defined by the transport provider that provides the transport service.

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).

host byte order
UX The implementation-dependent byte order supported by the local host machine (see **network byte order**). Functions are provided to convert 16 and 32-bit values between network and host byte order (see *htonl*( )).

initiator
An entity that initiates a connect request.

network byte order
UX The byte order in which the most significant byte of a multibyte integer value is transmitted first. This byte order is the standard byte order for Internet protocols.

network host database
UX A database whose entries define the names and network addresses of host machines. See *gethostent*( ).

network net database

UX A database whose entries define the names and network numbers of networks. See *getnetent*( ).

network protocol database

UX A database whose entries define the names and protocol numbers of protocols. See *getprotoent*( ).

network service database

UX 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.

responder

An entity with whom an initiator wishes to establish a transport connection.

socket

UX A communications endpoint associated with a file descriptor that provides communications services using a specified communications protocol.

synchronous mode

The mode of execution in which transport service functions wait for specific asynchronous events to occur before returning control to the user.

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.

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.

Index