CAE Specification

Transport Provider Interface (TPI)

The Open Group

CAE Specification

Transport Provider Interface (TPI)

ISBN: 1-85912-102-0
Document Number: C615

Published in the U.K. by The Open Group, February 1997.

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

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

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

Chapter	**1**	**Introduction**	**1**
	1.1	STREAMS-based Transport Provider Interface	1
	1.2	How TPI Works	2
	1.3	Overview of Error Handling Capabilities	3
	1.3.1	Non-fatal Errors	3
	1.3.2	Fatal Errors	3
	1.4	Rules for Transport Service Interface Sequence of Primitives	4
	1.5	Rules for Precedence of TPI Primitives on a Stream	5
	1.6	Rules for Flushing Queues	5
Chapter	**2**	**Transport Primitives**	**7**
Chapter	**3**	**Allowable Sequence of TPI Primitives**	**9**
	3.1	State Table	10
	3.2	Variables	11
	3.3	Outgoing Events	11
	3.4	Incoming Events	12
	3.5	Transport Service State Tables	13
Chapter	**4**	**Transport Primitive Precedence**	**17**
Chapter	**5**	**TPI Message Formats**	**19**
		T_ADDR_ACK	20
		T_ADDR_REQ	21
		T_BIND_ACK	22
		T_BIND_REQ	24
		T_CONN_CON	26
		T_CONN_IND	27
		T_CONN_REQ	28
		T_CONN_RES	30
		T_DATA_IND	33
		T_DATA_REQ	34
		T_DISCON_IND	36
		T_DISCON_REQ	37
		T_ERROR_ACK	39
		T_EXDATA_IND	40
		T_EXDATA_REQ	41
		T_INFO_ACK	42
		T_INFO_REQ	44
		T_OK_ACK	45
		T_OPTMGMT_ACK	46
		T_OPTDATA_IND	48

T_OPTDATA_REQ 50
T_OPTMGMT_REQ 52
T_ORDREL_IND 54
T_ORDREL_REQ 55
T_UDERROR_IND 56
T_UNBIND_REQ 57
T_UNITDATA_IND 58
T_UNITDATA_REQ 59

Appendix A Connection Acceptance **61**
A.1 Accepting Incoming Connections 61
A.2 The Common Single Type Model Implementation 62
A.3 Possible Multiple Type Model Implementation Methodologies 63

Glossary **65**

Index **67**

List of Figures

1-1 Example of a Stream from a User to a Transport Provider 2

List of Tables

2-1 Transport Service Primitives 7
3-1 Kernel Level Transport Interface States 10
3-2 State Table Variables 11
3-3 Kernel Level Transport Interface Outgoing Events 11
3-4 Kernel Level Transport Interface Incoming Events 12
3-5 Initialization State Table 13
3-6 Data Transfer State Table for Connection Oriented Service 14
3-7 Data Transfer State Table for Connectionless Service 15
4-1 Stream Write Queue Precedence Table 17
4-2 Stream Read Queue Precedence Table 18

Preface

The Open Group

The Open Group is an international open systems organisation that is leading the way in creating the infrastructure needed for the development of network-centric computing and the information superhighway. Formed in 1996 by the merger of the X/Open Company and the Open Software Foundation, The Open Group is supported by most of the world's largest user organisations, information systems vendors and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to assist user organisations, vendors and suppliers in the development and implementation of products supporting the adoption and proliferation of open systems.

With more than 300 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritising and communicating customer requirements to vendors
- conducting research and development with industry, academia and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the X/Open brand that designates vendor products which conform to X/Open Product Standards
- promoting the benefits of open systems to customers, vendors and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trade mark on behalf of the industry.

The X/Open Process

This description is used to cover the whole Process developed and evolved by X/Open. It includes the identification of requirements for open systems, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The X/Open brand logo is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the X/Open brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical literature, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our product standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. In addition, they can demonstrate product compliance through the X/Open brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organisations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

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

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation — programmer's guides, user manuals, and so on — relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Programme. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism 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 a technical activity.

Versions and Issues of Specifications

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

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.

- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

This Document

The Transport Provider Interface (TPI) defines an interface for drivers that provide transport services. The TPI specifies the set of messages and their formats which the driver must generate or process.

This specification has been developed from the original TPI Specification which was generated by UNIX International (UI). UI intellectual property rights were subsequently acquired by UNIX System Laboratories (USL), who in turn were later acquired by Novell Inc. See also the Acknowledgements page.

Intended Audience

This specification assumes the reader is familiar with OSI Reference Model terminology, OSI transport services and STREAMS.

Structure

The structure of this specifications is:

- Chapter 1, **Introduction**, describes the transport provider interface (TPI) as it is defined in the STREAMS environment
- Chapter 2, **Mapping to OSI**, describes the mapping of transport primitives to OSI
- Chapter 3, **Allowable Sequence of TPI Primitives**, describes the possible events and states for TPI
- Chapter 4, **Transport Primitive Precedence**, defines transport primitives precedence for stream *write* and *read* queues
- Chapter 5, **Message Formats**, gives the man-page definitions for the TPI message formats (structures)
- Appendix A, **Connection Acceptance**, offers background information to explain connection acceptance under existing common implementations, to help understanding of existing implementations and design of new ones.

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.
- 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

Motif®, OSF/1® and UNIX® are registered trade marks and the ‘‘X Device’’™ and The Open Group™ are trade marks of The Open Group.

Acknowledgements

The original TPI Specification was produced by UNIX International (UI). UI intellectual property rights subsequently passed to UNIX System Laboratories (USL), who in turn were acquired by Novell Inc. The Open Group acknowledges Novell's contribution of their TPI 2.01 specification as the base document from which this TPI specification was developed.

Referenced Documents

The following documents are referenced in this specification:

TPI-SMD
: UNIX Press (A Prentice Hall Title) book "STREAMS Modules and Drivers", published 1992, ISBN 0-13-066879-6.

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

Chapter 1

Introduction

1.1 STREAMS-based Transport Provider Interface

The Transport Provider Interface (TPI) is an interface for drivers that provide transport services. The TPI defines the set of messages and their formats that the driver must generate/process.

This chapter introduces the STREAMS-based Transport Provider Interface (TPI). TPI is a service interface that maps to strategic levels of the Open Systems Interconnection (OSI) Reference Model. TPI supports the services of the Transport Layer for connection-mode and connectionless-mode services.

One advantage to using TPI is its ability to hide implementation details of a particular service from the consumer of the service. This enables system programmers to develop software independent of the particular protocol that provides a specific service.

This chapter focuses on TPI as it is defined within the STREAMS environment. Although there are no formal standards for a STREAMS environment, extensive descriptions STREAMS and STREAMS programming can be found in the referenced document **TPI_SMD**.

1.2 How TPI Works

TPI defines a message interface to a transport provider implemented under STREAMS. A user communicates to a transport provider via a full duplex path known as a *stream* (see Figure 1-1). This *stream* provides a mechanism in which messages may be passed to the transport provider from the transport user and vice versa.

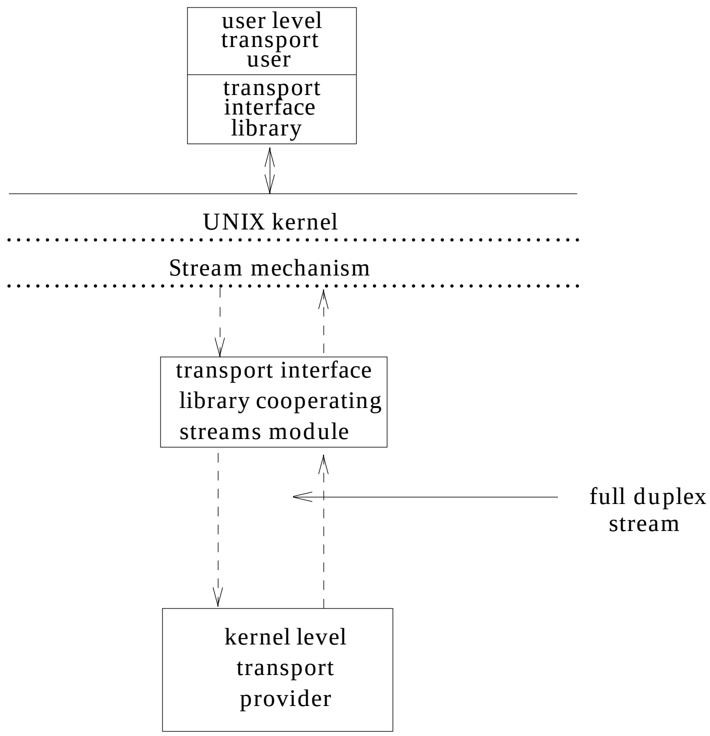

Figure 1-1 Example of a Stream from a User to a Transport Provider

The STREAMS messages that are used to communicate between the transport user and the transport provider may have one of the following formats:

- A **M_PROTO** message block followed by zero or more **M_DATA** message blocks. The **M_PROTO** message block contains the type of transport service primitive and all the relevant arguments associated with the primitive. The **M_DATA** blocks contain transport user data associated with the transport service primitive.
- One **M_PCPROTO** message block containing the type of transport service primitive and all the relevant arguments associated with the primitive.
- One or more **M_DATA** message blocks containing transport user data.
- One **M_ERROR** message block indicating that an unrecoverable error has occurred.
- One **M_FLUSH** message block indicating that queued requests should be discarded.

Chapter 5 contains descriptions of the transport primitives which define both a connection-mode and connectionless-mode transport service. There are also primitives that pertain to both transport modes.

For each type of transport service, two types of primitives exist:

- Primitives which originate from the transport user.

These make requests to the transport provider or respond to an event of the transport provider.

- Primitives which originate from the transport provider.

  These are either confirmations of a request or are indications to the transport user that an event has occurred.

For the connection-mode transport service, a connection is associated with a single stream and, except while processing inbound connections, a stream will have at most one connection associated with it.

Chapter 2 lists the primitive types along with the mapping of those primitives to the STREAMS message types and the transport primitives of the ISO IS 8072 and IS 8072/DAD transport service definitions (see referenced documents). The format of these primitives and the rules governing the use of them are described in Chapter 3.

1.3 Overview of Error Handling Capabilities

There are two error handling facilities available to the transport user:

- one to handle non-fatal errors
- one to handle fatal errors.

1.3.1 Non-fatal Errors

The non-fatal errors are those that a transport user can correct, and are reported in the form of an error acknowledgment to the appropriate primitive in error. Only those primitives which require acknowledgments may generate a non-fatal error acknowledgment. These acknowledgments always report a syntactical error in the specified primitive when the transport provider receives the primitive. The primitive descriptions above define those primitives and rules regarding the acknowledgment of them. These errors are reported to the transport user via the T_ERROR_ACK primitive, and give the transport user the option of reissuing the transport service primitive that caused the error. The T_ERROR_ACK primitive also indicates to the transport user that no action was taken by the transport provider on receipt of the primitive which caused the error.

These errors do not change the state of the transport service interface as seen by the transport user. The state of the interface after the issuance of a T_ERROR_ACK primitive should be the same as it was before the transport provider received the interface primitive that was in error.

The allowable errors that can be reported on the receipt of a transport initiated primitive are presented in the description of the appropriate primitives.

1.3.2 Fatal Errors

Fatal errors are those which can not be corrected by the transport user, or those errors which result in an uncorrectable error in the interface or in the transport provider.

The most common of these errors are listed under the appropriate primitives. The transport provider should issue fatal errors only if the transport user can not correct the condition which caused the error or if the transport provider has no means of reporting a transport user correctable error. If the transport provider detects an uncorrectable non-protocol error internal to the transport provider, the provider should issue a fatal error to the user.

Fatal errors are indicated to the transport user via the STREAMS message type **M_ERROR** with an appropriate UNIX system error. **EPROTO** should be used if the user has broken the TPI protocol. The message **M_ERROR** will result in the failure of all the operating system service routines on the **stream**. The user must then close the stream and, if required, attempt to open a new stream to the provider. Note that some providers may reject the ''open'' if, for example, the reason for the fatal error is that the provider has been shut down.

1.4 Rules for Transport Service Interface Sequence of Primitives

The allowable sequence of primitives are described in the state diagrams and tables in Chapter 3, for both the connection-mode and connectionless-mode transport services. The following are rules regarding the maintenance of the state of the interface:

- It is the responsibility of the transport provider to keep record of the state of the interface as viewed by the transport user.
- The state of the endpoint known by the transport user may differ from that kept by the provider (and returned in T_INFO_ACK messages) if there are messages queued on the read or write side of the stream.
- The transport provider must not generate a primitive that is illegal in the current state of the endpoint.
- The uninitialized state of a **stream** is the initial and final state, and it must be bound (see the T_BIND_REQ primitive, *T_BIND_REQ* on page 24) before the transport provider may view it as an active **stream**.
- If the transport provider sends a **M_ERROR** upstream, it should also drop any further messages received on its write side of the **stream**.

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

- A transport connection release procedure can be initiated at any time during the transport connection establishment or data transfer phase.
- The state tables for the connection-mode transport service providers include the management of the sequence numbering when a transport provider sends multiple T_CONN_IND requests without waiting for the response of the previously sent indication. It is the responsibility of the transport providers not to change state until all the indications have been responded to. Therefore the provider should remain in the TS_WRES_CIND state while there are any outstanding connect indications pending response. The provider should change state appropriately when all the connect indications have been responded to.
- The state of a transport service interface of a **stream** may only be transferred to another **stream** when it is indicated in a T_CONN_RES primitive. The following rules then apply to the cooperating **streams**:
  - — The **stream** which is to accept the current state may be unbound, or bound but not connected to a peer.
  - — The user transferring the current state of a **stream** must have correct permissions for the use of the protocol address bound to the accepting **stream**.
  - — The **stream** which transfers the state of the transport interface must be placed into an appropriate state after the completion of the transfer.

1.5 Rules for Precedence of TPI Primitives on a Stream

The following rules apply to the precedence of transport interface primitives with respect to their position on a **stream**:

- The transport provider has responsibility for determining precedence on its *stream write* queue, as described in the rules in Chapter 4. This section specifies the rules for precedence for both the connection-mode and connectionless-mode transport services.
- The transport user has responsibility for determining precedence on its *stream read* queue, as described in the rules in Chapter 4.
- All primitives on the **stream** are assumed to be placed on the queue in the correct sequence as defined above.

Note: The **stream** queue which contains the transport user initiated primitives is referred to as the *stream write* queue. The **stream** queue which contains the transport provider initiated primitives is referred to as the *stream read* queue.

The following rule applies only to the connection-mode transport services:

- There is no guarantee of delivery of user data once a T_DISCON_REQ primitive has been issued.

1.6 Rules for Flushing Queues

The following rules pertain to flushing the stream queues. No other flushes should be needed to keep the queues in the proper condition.

- The transport providers must be aware that they will receive **M_FLUSH** messages from upstream. These flush requests are issued to ensure that the providers receive certain messages and primitives. It is the responsibility of the providers to act appropriately as deemed necessary by the providers.
- The transport provider must send up a **M_FLUSH** message to flush both the read and write queues after receiving a successful T_UNBIND_REQ message and before issuing the T_OK_ACK primitive.

The following rules pertain only to the connection-mode transport providers.

- If the interface is in the TS_DATA_XFER, TS_WIND_ORDREL or TS_WACK_ORDREL state, the transport provider must send up a **M_FLUSH** message to flush both the read and write queues before sending up a **T_DISCON_IND**.
- If the interface is in the TS_DATA_XFER, TS_WIND_ORDREL or TS_WACK_ORDREL state, the transport provider must send up a **M_FLUSH** message to flush both the read and write queues after receiving a successful T_DISCON_REQ message and before issuing the T_OK_ACK primitive.

Chapter 2

Transport Primitives

The following table lists the TPI primitives with a brief description, and gives the streams message type.

Transport Primitives	Description	Stream Message Types
T_BIND_REQ	Bind Protocol Address Request	**M_PROTO**
T_BIND_ACK	Bind Protocol Address Acknowledgement	**M_PCPROTO**
T_CONN_REQ	Connection Request	**M_PROTO**
T_CONN_IND	Connection Indication	**M_PROTO**
T_CONN_RES	Connection Response	**M_PROTO**
T_CONN_CON	Connection Confirm	**M_PROTO**
T_DATA_REQ	Data Request	**M_PROTO**
T_DATA_IND	Data Indication	**M_PROTO**
T_DISCON_REQ	Disconnect Request	**M_PROTO**
T_DISCON_IND	Disconnect Indication	**M_PROTO**
T_ERROR_ACK	Error Acknowledgement	**M_PCPROTO**
T_EXDATA_REQ	Expedited Data Request	**M_PROTO**
T_EXDATA_IND	Expedited Data Indication	**M_PROTO**
T_INFO_REQ	Transport Protocol Parameters Request	**M_PCPROTO**
T_INFO_ACK	Transport Protocol Parameters Acknowledgement	**M_PCPROTO**
T_OK_ACK	Success Acknowledgement	**M_PCPROTO**
T_OPTDATA_REQ	Data Request with Options	**M_PROTO**
T_OPTDATA_IND	Data Indication with Options	**M_PROTO**
T_OPTMGMT_REQ	Options Management Request	**M_PROTO**
T_OPTMGMT_ACK	Options Management Acknowledgement	**M_PCPROTO**
T_ORDREL_REQ	Orderly Release Request	**M_PROTO**
T_ORDREL_IND	Orderly Release Indication	**M_PROTO**
T_UDERROR_IND	Unitdata Error Indication	**M_PROTO**
T_UNBIND_REQ	Unbind Protocol Address Request	**M_PROTO**
T_UNITDATA_REQ	Unitdata Request	**M_PROTO**
T_UNITDATA_IND	Unitdata Indication	**M_PROTO**

Table 2-1 Transport Service Primitives

Chapter 3

Allowable Sequence of TPI Primitives

The following tables describe the possible events that may occur on the interface and the possible states as viewed by the transport user that the interface may enter due to an event. The events map directly to the transport service interface primitives as described in Chapter 1.

3.1 State Table

Possible States			
State		Description	Service Type
Name	Abbreviation		
TS_UNBND	*sta_0*	unbound	T_COTS, T_COTS_ORD, T_CLTS
TS_WACK_BREQ	*sta_1*	awaiting acknowledgment of T_BIND_REQ	T_COTS, T_COTS_ORD, T_CLTS
TS_WACK_UREQ	*sta_2*	awaiting acknowledgment of T_UNBIND_REQ	T_COTS, T_COTS_ORD,
TS_IDLE	*sta_3*	idle - no connection	T_COTS, T_COTS_ORD, T_CLTS
TS_WACK_OPTREQ	*sta_4*	awaiting acknowledgment of T_OPTMGMT_REQ	T_COTS, T_COTS_ORD, T_CLTS
TS_WACK_CREQ	*sta_5*	awaiting acknowledgment of T_CONN_REQ	T_COTS, T_COTS_ORD
TS_WCON_CREQ	*sta_6*	awaiting confirmation of T_CONN_REQ	T_COTS, T_COTS_ORD
TS_WRES_CIND	*sta_7*	awaiting response of T_CONN_IND	T_COTS, T_COTS_ORD
TS_WACK_CRES	*sta_8*	awaiting acknowledgment of T_CONN_RES	T_COTS, T_COTS_ORD
TS_DATA_XFER	*sta_9*	data transfer	T_COTS, T_COTS_ORD
TS_WIND_ORDREL	*sta_10*	awaiting T_ORDREL_IND	T_COTS_ORD
TS_WREQ_ORDREL	*sta_11*	awaiting T_ORDREL_REQ	T_COTS_ORD
TS_WACK_DREQ6	*sta_12*	awaiting acknowledgment of T_DISCON_REQ	T_COTS, T_COTS_ORD
TS_WACK_DREQ7	*sta_13*	awaiting acknowledgment of T_DISCON_REQ	T_COTS, T_COTS_ORD
TS_WACK_DREQ9	*sta_14*	awaiting acknowledgment of T_DISCON_REQ	T_COTS, T_COTS_ORD
TS_WACK_DREQ10	*sta_15*	awaiting acknowledgment of T_DISCON_REQ	T_COTS, T_COTS_ORD
TS_WACK_DREQ11	*sta_16*	awaiting acknowledgment of T_DISCON_REQ	T_COTS, T_COTS_ORD

sta_0, *sta_1*, etc. are convenient abbreviations used in the state tables later in this Chapter.

Table 3-1 Kernel Level Transport Interface States

3.2 Variables

The following table describes the variables used in the state tables.

Variable	Description
q	queue pair pointer of current **stream**
rq	queue pair pointer of responding **stream** as described in the T_CONN_RES primitive
outcnt	counter for the number of outstanding connection indications not responded to by the transport user

Table 3-2 State Table Variables

3.3 Outgoing Events

The following outgoing events are those which are initiated from the transport service user. They either make requests of the transport provider or respond to an event of the transport provider.

EVENT	DESCRIPTION	SERVICE TYPE
bind_req	bind request	T_COTS, T_COTS_ORD, T_CLTS
unbind_req	unbind request	T_COTS, T_COTS_ORD, T_CLTS
optmgmt_req	options mgmt request	T_COTS, T_COTS_ORD, T_CLTS
conn_req	connection request	T_COTS, T_COTS_ORD
conn_res	connection response	T_COTS, T_COTS_ORD
discon_req	disconnect request	T_COTS, T_COTS_ORD
data_req	data request	T_COTS, T_COTS_ORD
exdata_req	expedited data request	T_COTS, T_COTS_ORD
optdata_req	data request with options	T_COTS, T_COTS_ORD
ordrel_req	orderly release request	T_COTS_ORD
unitdata_req	unitdata request	T_CLTS

Table 3-3 Kernel Level Transport Interface Outgoing Events

3.4 Incoming Events

The following incoming events are those which are initiated from the transport provider. They are either confirmations of a request or are indications to the transport user that an event has occurred.

EVENT	DESCRIPTION	SERVICE TYPE
bind_ack	bind acknowledgment	T_COTS, T_COTS_ORD, T_CLTS
optmgmt_ack	options mgmt acknowledgment	T_COTS, T_COTS_ORD, T_CLTS
error_ack	error acknowledgment	T_COTS, T_COTS_ORD, T_CLTS
ok_ack1	ok acknowledgment outcnt == 0	T_COTS, T_COTS_ORD, T_CLTS
ok_ack2	ok acknowledgment outcnt == 1, q == rq	T_COTS, T_COTS_ORD,
ok_ack3	ok acknowledgment outcnt == 1, q	T_COTS, T_COTS_ORD, = rq
ok_ack4	ok acknowledgment outcnt > 1	T_COTS, T_COTS_ORD,
conn_ind	connection indication	T_COTS, T_COTS_ORD
conn_con	connection confirmation	T_COTS, T_COTS_ORD
data_ind	data indication	T_COTS, T_COTS_ORD
exdata_ind	expedited data indication	T_COTS, T_COTS_ORD
optdata_ind	data indication with options	T_COTS, T_COTS_ORD
ordrel_ind	orderly release indication	T_COTS_ORD
discon_ind1	disconnect indication outcnt == 0	T_COTS, T_COTS_ORD
discon_ind2	disconnect indication outcnt == 1	T_COTS, T_COTS_ORD
discon_ind3	disconnect indication outcnt > 1	T_COTS, T_COTS_ORD
pass_conn	pass connection	T_COTS, T_COTS_ORD
unitdata_ind	unitdata indication	T_CLTS
uderror_ind	unitdata error indication	T_CLTS

Table 3-4 Kernel Level Transport Interface Incoming Events

3.5 Transport Service State Tables

The next three tables describe the possible next states the interface may enter, given a current state and event.

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. The transport provider must take the specific actions in the order specified in the state table.

	STATE				
EVENT	**TS_UNBND** *sta_0*	**TS_WACK_BREQ** *sta_1*	**TS_WACK_UREQ** *sta_2*	**TS_IDLE** *sta_3*	**TS_WACK_OPTREQ** *sta_4*
bind_req	*sta_1*				
unbind_req				*sta_2*	
optmgmt_req	*sta_4* [5]			*sta_4* [5]	
bind_ack		*sta_3*	[1]		
optmgmt_ack					*sta_3*
error_ack		*sta_0*	*sta_3*		*sta_3*
ok_ack1			*sta_0*		

[1] outcnt = 0

[5] return to previous state

sta_0, sta_1, etc. are convenient abbreviations for different states — see Table 3-1 on page 10.

Table 3-5 Initialization State Table

	STATE													
	TS_ UNBND	**TS_ IDLE**	**TS_ WACK_ CREQ**	**TS_ WCON_ CREQ**	**TS_ WRES_ CIND**	**TS_ WACK_ CRES**	**TS_ DATA_ XFER**	**TS_ WIND_ ORDREL**	**TS_ WREQ_ ORDREL**	**TS_ WACK_ DREQ6**	**TS_ WACK_ DREQ7**	**TS_ WACK_ DREQ9**	**TS_ WACK_ DREQ10**	**TS_ WACK_ DREQ11**
EVENT	*sta_0*	*sta_3*	*sta_5*	*sta_6*	*sta_7*	*sta_8*	*sta_9*	*sta_10* **	*sta_11* **	*sta_12*	*sta_13*	*sta_14*	*sta_15*	*sta_16*
conn_req		*sta_5*												
conn_res					*sta_8*									
discon_req				*sta_12*	*sta_13*		*sta_14*	*sta_15*	*sta_16*					
data_req							*sta_9*		*sta_11*					
exdata_req							*sta_9*		*sta_11*					
** ordrel_req							*sta_10*		*sta_3*					
conn_ind		*sta_7* *[2]*			*sta_7* *[2]*									
conn_con				*sta_9*										
data_ind							*sta_9*	*sta_10*						
exdata_ind							*sta_9*	*sta_10*						
** ordrel_ind							*sta_11*	*sta_3*						
discon_ind1				*sta_3*			*sta_3*	*sta_3*	*sta_3*					
discon_ind2					*sta_3*	*[3]*								
discon_ind3					*sta_3*	*[3]*								
optmgmt_req		*sta_4* *[5]*	*sta_4* *[5]*		*sta_4* *[5]*	*sta_4* *[5]*		*sta_4* *[5]*	*sta_4* *[5]*	*sta_4* *[5]*				
error_ack			*sta_3*			*sta_7*				*sta_6*	*sta_7*	*sta_9*	*sta_10*	*sta_11*
ok_ack1			*sta_6*							*sta_3*		*sta_3*	*sta_3*	*sta_3*
ok_ack2						*sta_9* *[3]*					*sta_3* *[3]*			
ok_ack3						*sta_3* *[3] [4]*					*sta_3* *[3]*			
ok_ack4						*sta_7* *[3] [4]*					*sta_7* *[3]*			
pass_conn	*sta_9*	*sta_9*												

** Only supported if service is type T_COTS_ORD

[2] outcnt = outcnt + 1

[3] outcnt = outcnt = 1

[4] pass connection to queue as indicated in the T_CON_RES primitive

[5] return to previous state.

sta_0, *sta_3*, etc. are convenient abbreviations for different states — see Table 3-1 on page 10.

Table 3-6 Data Transfer State Table for Connection Oriented Service

EVENT	STATE
unitdata_req	TS_IDLE
unitdata_ind	TS_IDLE
uderror_ind	TS_IDLE

Table 3-7 Data Transfer State Table for Connectionless Service

Chapter 4

Transport Primitive Precedence

Table 4-1 describes the precedence of the transport primitives for **stream** write queues, and Table 4-2 describes this for **stream** read queues.

Note: The **stream** queue which contains the transport user initiated primitives is referred to as the stream *write queue*. The **stream** queue which contains the transport provider initiated primitives is referred to as the stream *read queue*.

Y \ X	*t_conn_req*	*t_conn_res*	*t_discon_req*	*t_data_req*	*t_exdata_req*	*t_bind_req*	*t_unbind_req*	*t_info_req*	*t_unitdata_req*	*t_optmgmt_req*	*t_ordrel_req*
t_conn_req			4								
t_conn_res			3								
t_discon_req											
t_data_req			5	1	2						1
t_exdata_req			5	1	1						1
t_bind_req											
t_unbind_req											
t_info_req											
t_unitdata_req									1		
t_optmgmt_req											
t_ordrel_req			5								

Key

blank: not applicable / queue should be empty

1 : X has no precedence over Y

2 : X has precedence over Y

3 : X has precedence over Y
and Y must be removed

4 : X has precedence over Y and
both X and Y must be removed

5 : X may have precedence over Y
(choice of user) and if X does, then
it is the same as 3

Table 4-1 Stream Write Queue Precedence Table

Y \ X	*t_conn_ind*	*t_conn_con*	*t_discon_ind*	*t_data_ind*	*t_exdata_ind*	*t_info_ack*	*t_bind_ack*	*t_error_ack*	*t_ok_ack*	*t_unitdata_ind*	*t_uderror_ind*	*t_optmgmt_ack*	*t_ordrel_ack*
t_conn_ind			4										
t_conn_con			3	1	1								
t_discon_ind	1							2	2				
t_data_ind			5	1	2				1				1
t_exdata_ind			5	1	1				1				1
t_info_ack													
t_bind_ack	1												
t_error_ack	1	1	1	1	1								
t_ok_ack	1	1	1	1	1								
t_unitdata_ind								2		1	2	2	
t_uderror_ind								1		1	1	1	
t_optmgmt_ack	1									1	1		
t_ordrel_ack	1		5					2	2				

Key

blank: not applicable / queue should be empty

1 : X has no precedence over Y

2 : X has precedence over Y

3 : X has precedence over Y
and Y must be removed

4 : X has precedence over Y and
both X and Y must be removed

5 : X may have precedence over Y
(choice of user) and if X does, then
it is the same as 3

Table 4-2 Stream Read Queue Precedence Table

Chapter 5

TPI Message Formats

SYNOPSIS

```
include <sys/types.h>
include <sys/ddi.h>
```

DESCRIPTION

The Transport Provider Interface (TPI) Message Formats define the message formats (structures) used by the service primitives. These are classified as connection-mode, connectionless-mode, or both. They are further classified as being either user-originated or provider-originated.

Two **types** are used to build the TPI primitives. The normative definitions of **t_scalar_t** and **t_uscalar_t** are to be found in the Networking Services Specification (see the referenced **XNS** specification), but are repeated here for informational purposes.

t_scalar_t and **t_uscalar_t** are, respectively, a signed and an unsigned opaque integral type of equal length of at least 32 bits[1].

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

NAME

T_ADDR_ACK - Protocol Address Acknowledgment

SYNOPSIS

This message consists of one M_PCPROTO message block formatted as follows:

```
struct T_addr_ack {
    t_scalar_t   PRIM_type;      /* Always T_ADDR_ACK */
    t_scalar_t   LOCADDR_length;
    t_scalar_t   LOCADDR_offset;
    t_scalar_t   REMADDR_length;
    t_scalar_t   REMADDR_offset;
};
```

DESCRIPTION

This primitive indicates to the transport user the local and remote protocol addresses currently associated with the transport endpoint.

PARAMETERS

The fields of this message have the following meanings:

PRIM_type
: the primitive type.

LOCADDR_length
: the length of the local address associated with the transport endpoint.

LOCADDR_offset
: the offset from the beginning of the M_PCPROTO message block where the local address begins.

REMADDR_length
: the length of the remote address associated with the transport endpoint.

REMADDR_offset
: the offset from the beginning of the M_PCPROTO message block where the remote address begins.

The proper alignment of the addresses in the M_PCPROTO message block is not guaranteed.

RULES

If the transport endpoint is not bound to a local address, the *LOCADDR_length* field is set to 0.

If the transport endpoint is not associated with a remote address, the *REMADDR_length* field is set to 0.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport provider.

NAME

T_ADDR_REQ - Get Protocol Address Request

SYNOPSIS

This message consists of a M_PCPROTO message block formatted as follows:

```
struct T_addr_req {
    t_scalar_t   PRIM_type;      /* Always T_ADDR_REQ */
};
```

DESCRIPTION

This primitive requests the transport provider to return the local and remote protocol addresses currently associated with the transport endpoint.

PARAMETERS

PRIM_type
indicates the primitive type.

Note that the T_ADDR_REQ and T_ADDR_ACK primitives have no effect on the state of the transport provider and do not appear in the state tables.

RULES

This primitive requires the transport provider to generate one of the following acknowledgments on receipt of the primitive and that the transport user wait for the acknowledgment before issuing any other primitives:

Successful
Acknowledgment of the primitive via T_ADDR_ACK

Non-fatal errors
These errors will be indicated via T_ERROR_ACK.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport user.

NAME

T_BIND_ACK - Bind Protocol Address Acknowledgment

SYNOPSIS

This message consists of one M_PCPROTO message block formatted as follows:

```
struct T_bind_ack {
    t_scalar_t    PRIM_type;      /* Always T_BIND_ACK */
    t_scalar_t    ADDR_length;
    t_scalar_t    ADDR_offset;
    t_uscalar_t   CONIND_number;
};
```

DESCRIPTION

This primitive indicates to the transport user that the specified protocol address has been bound to the stream, that the specified number of connect indications are allowed to be queued by the transport provider for the specified protocol address, and that the stream associated with the specified protocol address has been activated.

PARAMETERS

PRIM_type
indicates the primitive type.

ADDR_length
is the length of the protocol address that was bound to the stream.

ADDR_offset
is the offset from the beginning of the M_PCPROTO block where the protocol address begins.

CONIND_number
is the accepted number of connect indications allowed to be outstanding by the transport provider for the specified protocol address.

Note that this field does not apply to connectionless transport providers.

The proper alignment of the address in the M_PCPROTO message block is not guaranteed.

RULES

The following rules apply to the binding of the specified protocol address to the stream:

- If the *ADDR_length* field in the T_BIND_REQ / O_T_BIND_REQ primitive is 0, then the transport provider is to assign a transport protocol address to the user.
- The transport provider is to bind the transport protocol address as specified in the T_BIND_REQ / O_T_BIND_REQ primitive.
- If the transport provider cannot bind the specified address, it may assign another address to the user if the primitive O_T_BIND_REQ was used. In this case, it is the transport user's responsibility to check the protocol address returned in the T_BIND_ACK primitive to see if it is the same as the one requested, and take appropriate action. If T_BIND_REQ was used, the provider should return an error.

The following rules apply to negotiating the *CONIND_number* argument:

- The returned value must be less than or equal to the corresponding requested number as indicated in the T_BIND_REQ / O_T_BIND_REQ primitive.
- If the requested value is greater than zero, the returned value must also be greater than zero.

- Only one stream that is bound to the indicated protocol address may have a negotiated accepted number of maximum connect requests greater than zero. If a O_T_BIND_REQ primitive specifies a value greater than zero, but another stream has already bound itself to the given protocol address with a value greater than zero, the transport provider should assign another protocol address to the user.
- If a stream with *CONIND_number* greater than zero is used to accept a connection, the stream will be found busy during the duration of that connection and no other stream may be bound to that protocol address with a *CONIND_number* greater than zero. This will prevent more than one stream bound to the identical protocol address from accepting connect indications.
- A stream requesting a *CONIND_number* of zero should always be valid. This indicates to the transport provider that the stream is to be used to request connections only.
- A stream with a negotiated *CONIND_number* greater than zero may generate connect requests or accept connect indications.

ERRORS

If the above rules result in an error condition, then the transport provider must issue an T_ERROR_ACK primitive to the transport user specifying the error as defined in the description of the T_BIND_REQ / O_T_BIND_REQ primitive.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport provider.

NAME

T_BIND_REQ / O_T_BIND_REQ - Bind Protocol Address Request

SYNOPSIS

These messages consist of one M_PROTO message block formatted as follows:

```
struct T_bind_req {
    t_scalar_t    PRIM_type;      /* Always T_BIND_REQ */
    t_scalar_t    ADDR_length;
    t_scalar_t    ADDR_offset;
    t_uscalar_t   CONIND_number;
};
```

DESCRIPTION

These primitives request that the transport provider bind a protocol address to the stream, negotiate the number of connect indications allowed to be outstanding by the transport provider for the specified protocol address, and activate the stream associated with the protocol address.

- Note that a stream is viewed as active when the transport provider may receive and transmit TPDUs (transport protocol data units) associated with the stream.

PARAMETERS

PRIM_type
indicates the primitive type.

ADDR_length
is the length of the protocol address to be bound to the stream.

ADDR_offset
is the offset from the beginning of the M_PROTO block where the protocol address begins.

Note that all lengths, offsets, and sizes in all structures refer to the number of bytes.

CONIND_number
is the requested number of connect indications allowed to be outstanding by the transport provider for the specified protocol address.

Note that the *CONIND_number* should be ignored by those providing a connectionless transport service.

Also note that if the number of outstanding connect indications equals *CONIND_number*, the transport provider need not discard further incoming connect indications, but may choose to queue them internally until the number of outstanding connect indications drops below *CONIND_number*.

The proper alignment of the address in the M_PROTO message block is not guaranteed. The address in the M_PROTO message block is however, aligned the same as it was received from the transport user.

RULES

For rules governing the requests made by these primitives, see the T_BIND_ACK primitive.

These primitives require the transport provider to generate one of the following acknowledgments on receipt of the primitive, and the transport user must wait for the acknowledgment before issuing any other primitives:

Successful
Correct acknowledgment of the primitive is indicated via the T_BIND_ACK primitive.

Non-fatal errors

These errors will be indicated via the T_ERROR_ACK primitive described in reference **TPI-SMD**.

ERRORS

The allowable errors are as follows:

[TACCES]

This indicates that the user did not have proper permissions for the use of the requested address.

[TADDRBUSY]

This indicates that the requested address is in use. In other words, the transport user attempted to bind a protocol address to a second transport end point with a *CONIND_number* greater than zero. This error will only be returned for T_BIND_REQ. See T_BIND_ACK for the behavior of O_T_BIND_REQ in this instance.

[TBADADDR]

This indicates that the protocol address was in an incorrect format or the address contained invalid information. It is not intended to indicate protocol errors.

[TNOADDR]

This indicates that the transport provider could not allocate an address.

[TOUTSTATE]

The primitive would place the transport interface out of state.

[TSYSERR]

A system error has occurred and the UNIX system error is indicated in the primitive.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport user.

NAME

T_CONN_CON - Connection Confirm

SYNOPSIS

This message consists of one M_PROTO message block followed by zero or more M_DATA blocks if any user data is associated with the primitive. The format of the M_PROTO message block is as follows:

```
struct T_conn_con {
    t_scalar_t   PRIM_type;      /* Always T_CONN_CON         */
    t_scalar_t   RES_length;     /* Responding address length */
    t_scalar_t   RES_offset;
    t_scalar_t   OPT_length;
    t_scalar_t   OPT_offset;
};
```

DESCRIPTION

This primitive indicates to the user that a connect request has been confirmed on the specified responding address.

PARAMETERS

PRIM_type
identifies the primitive type.

RES_length
is the length of the responding address that the connection was accepted.

RES_offset
is the offset (from the beginning of the M_PROTO message block) where the responding address begins.

OPT_length
is the length of the confirmed options associated with the primitive.

OPT_offset
is the offset from the beginning of the M_PROTO message block) of the confirmed options associated with the primitive.

The proper alignment of the responding address and options in the M_PROTO message block is not guaranteed.

MODES

Only connection-mode.

ORIGINATOR

Transport provider.

NAME

T_CONN_IND - Connect Indication

SYNOPSIS

This message consists of one M_PROTO message block followed by zero or more M_DATA blocks if any user data is associated with the primitive. The format of the M_PROTO message block is as follows:

```
struct T_conn_ind {
    t_scalar_t   PRIM_type;       /* Always T_CONN_IND */
    t_scalar_t   SRC_length;
    t_scalar_t   SRC_offset;
    t_scalar_t   OPT_length;
    t_scalar_t   OPT_offset;
    t_scalar_t   SEQ_number;
};
```

DESCRIPTION

This primitive indicates to the transport user that a connect request to the user has been made by the user at the specified source address.

PARAMETERS

PRIM_type
identifies the primitive type.

SRC_length
is the length of the source address

SRC_offset
is the offset (from the beginning of the M_PROTO message block) where the source address begins.

OPT_length
is the length of the requested options associated with the primitive.

OPT_offset
is the offset (from the beginning of the M_PROTO message block) of the requested options associated with the primitive.

SEQ_number
should be a unique number other than -1 to identify the connect indication.

The proper alignment of the source address and options in the M_PROTO message block is not guaranteed.

MODES

Only connection-mode.

ORIGINATOR

Transport provider.

NAME

T_CONN_REQ **-** Connect Request

SYNOPSIS

This message consists of one M_PROTO message block followed by zero or more M_DATA blocks if any user data is specified by the transport user. The format of the M_PROTO message block is as follows:

```
struct T_conn_req {
    t_scalar_t   PRIM_type;      /* Always T_CONN_REQ */
    t_scalar_t   DEST_length;
    t_scalar_t   DEST_offset;
    t_scalar_t   OPT_length;
    t_scalar_t   OPT_offset;
};
```

DESCRIPTION

This primitive requests that the transport provider connect to the specified destination.

PARAMETERS

PRIM_type
: identifies the primitive type.

DEST_length
: is the length of the destination address

DEST_offset
: is the offset (from the beginning of the M_PROTO message block) where the destination address begins.

OPT_length
: is the length of the requested options associated with the primitive.

OPT_offset
: is the offset (from the beginning of the M_PROTO message block) of the requested options associated with the primitive.

The proper alignment of the destination address and options in the M_PROTO message block is not guaranteed. The destination address and options in the M_PROTO message block are however, aligned the same as they were received from the transport user.

Note: The information located by the defined structures may not be in the proper alignment in the message blocks, so the casting of structure definitions over these fields may produce incorrect results. It is advised that the transport providers supply exact format specifications for the appropriate information to the transport users.

RULES

This primitive requires the transport provider to generate one of the following acknowledgments on receipt of the primitive, and the transport user must wait for the acknowledgment before issuing any other primitives:

Successful
: Correct acknowledgment of the primitive is indicated via the T_OK_ACK primitive described in reference **TPI-SMD**.

Non-fatal errors

These errors will be indicated via the T_ERROR_ACK primitive described in reference **TPI-SMD**.

ERRORS

The allowable errors are as follows:

[TACCES]
This indicates that the user did not have proper permissions for the use of the requested address or options.

[TADDRBUSY]
The transport provider does not support multiple connections to the same destination address. This error indicates that a connection already exists for the requested destination.

[TBADADDR]
This indicates that the protocol address was in an incorrect format or the address contained invalid information. It is not intended to indicate protocol connection errors, such as an unreachable destination. Those error types are indicated via the T_DISCON_IND primitive.

[TBADDATA]
The amount of user data specified was invalid.

[TBADOPT]
This indicates that the options were in an incorrect format, or they contained invalid information.

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

[TOUTSTATE]
The primitive would place the transport interface out of state.

[TSYSERR]
A system error has occurred and the UNIX system error is indicated in the primitive.

MODES

Only connection-mode.

ORIGINATOR

Transport user.

NAME

T_CONN_RES - Connection Response

SYNOPSIS

This message consists of one M_PROTO message block followed by zero or more M_DATA blocks if any user data is specified by the transport user. The format of the M_PROTO message block is as follows:

```
struct T_conn_res {
    t_scalar_t   PRIM_type;     /* always T_CONN_RES     */
    t_uscalar_t  ACCEPTOR_id;   /* accepting endpoint ID */
    t_scalar_t   OPT_length;    /* options length        */
    t_scalar_t   OPT_offset;    /* options offset        */
    t_scalar_t   SEQ_number;    /* sequence number       */
};
```

DESCRIPTION

This primitive is sent by a transport user to the transport provider on a listening transport endpoint (hereafter, for brevity, referred to as the listener) on which the transport user received a T_CONN_IND. This primitive requests that the transport provider should accept the connection indication identified by *SEQ_number* on the response transport endpoint specified by *ACCEPTOR_id*.

PARAMETERS

PRIM_type
identifies the primitive type.

ACCEPTOR_id
identifies the transport provider endpoint which should be used to accept the connect request. The mapping of the contents of *ACCEPTOR_id* to the internal reference to a transport endpoint (often a pointer to a *streams* queue) is transport-provider defined. Some example mechanisms for using *ACCEPTOR_id*are*given*in Appendix A on page 61.

OPT_length
is the length of the responding options.

OPT_offset
is the offset from the beginning of the M_PROTO message block where the responding options begin.

SEQ_number
is the sequence number which identifies the connection being responded to.

The proper alignment of the options in the M_PROTO message block is not guaranteed. The options in the M_PROTO message block are, however, aligned the same as they were received from the transport user.

RULES

The following rules apply when the transport endpoint referenced by *ACCEPTOR_id* is not the same as the listener:

- If the endpoint referenced by *ACCEPTOR_id* is not bound at the time that the T_CONN_RES primitive is received by the transport provider, the transport provider will automatically bind that endpoint to the same protocol address as that to which the listener is bound.
- If the endpoint referenced by *ACCEPTOR_id* is already bound when the T_CONN_RES primitive was received by the transport provider, it must be bound to a protocol address with a *CONIND_number* of zero and must be in the TS_IDLE state.

In all cases, this primitive requires the transport provider to generate one of the following acknowledgments on receipt of the primitive, and the transport user wait for the acknowledgment before issuing any other primitives:

Successful
: Correct acknowledgment of the primitive is indicated via the T_OK_ACK primitive described in reference **TPI-SMD**.

Non-fatal errors
: These errors will be indicated via the T_ERROR_ACK primitive described in reference **TPI-SMD**.

ERRORS

The allowable errors are as follows:

[TACCES]
: This indicates that the user did not have proper permissions for the use of the options or response id.

[TBADADDR]
: The specified protocol address (the one bound to the endpoint referenced by *ACCEPTOR_id*) was in an incorrect format or contained illegal information.

[TBADDATA]
: The amount of user data specified was invalid.

[TBADF]
: This indicates that the response acceptor identifier was invalid.

[TBADOPT]
: This indicates that the options were in an incorrect format, or they contained invalid information.

[TBADSEQ]
: The sequence number specified in the primitive was incorrect or invalid.

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

[TOUTSTATE]
: The primitive would place the transport interface out of state.

[TPROVMISMATCH] This indicates that the response ACCEPTOR_Id does not identify a transport provider of the same type as the listener.

[TRESADDR]
: The transport provider requires both transport endpoints (that is, the one referenced by *ACCEPTOR_id* and the listener) to be bound to the same address.

[TRESQLEN]
: The endpoint referenced by *ACCEPTOR_id* was different from the listener, but was bound to a protocol address with a *CONIND_number* that is greater than zero.

[TSYSERR]
: A system error has occurred and the UNIX system error is indicated in the primitive.

MODES

Only connection-mode.

ORIGINATOR

Transport user.

NAME

T_DATA_IND - Data Indication

SYNOPSIS

This message consists of one M_PROTO message block followed by zero or more M_DATA message blocks where each M_DATA message block contains at least one byte of data. The format of the M_PROTO message block is as follows:

```
struct T_data_ind {
    t_scalar_t   PRIM_type;      /* Always T_DATA_IND */
    t_scalar_t   MORE_flag;
};
```

DESCRIPTION

This primitive indicates to the transport user that this message contains a transport interface data unit. One or more transport interface data units form a transport service data unit. This primitive has a mechanism which indicates the beginning and end of a transport service data unit. However, not all transport providers support the concept of a transport service data unit.

PARAMETERS

PRIM_type identifies the primitive type.

MORE_flag
: when greater than zero, indicates that the next T_DATA_IND primitive is also part of this transport service data unit.

RULES

If a TSDU spans multiple T_DATA_IND message blocks, then an ETSDU may be placed in between two T_DATA_IND message blocks. Once an ETSDU is started, then the ETSDU must be completed before any T_DATA_IND message blocks defining a TSDU is resumed.

MODES

Only connection-mode.

ORIGINATOR

Transport provider.

NAME

T_DATA_REQ - Data Request

SYNOPSIS

This message consists of one M_PROTO message block followed by zero or more M_DATA message blocks where each M_DATA message block contains zero or more bytes of data. The format of the M_PROTO message block is as follows:

```
struct T_data_req {
    t_scalar_t   PRIM_type;      /* Always T_DATA_REQ */
    t_scalar_t   MORE_flag;
};
```

DESCRIPTION

This primitive indicates to the transport provider that this message contains a transport interface data unit. One or more transport interface data units form a transport service data unit (TSDU).

Note that the maximum transport service data unit size allowed by the transport provider is indicated to the transport user via the T_INFO_ACK primitive.

This primitive has a mechanism which indicates the beginning and end of a transport service data unit. However, not all transport providers support the concept of a transport service data unit.

PARAMETERS

PRIM_type

identifies the primitive type.

MORE_flag

when greater than zero, indicates that the next T_DATA_REQ primitive is also part of this transport service data unit.

RULES

The transport provider must also recognize a message of one or more M_DATA message blocks without the leading M_PROTO message block as a T_DATA_REQ primitive. This message type will be initiated from the **write**(2) operating system service routine.

For example, on systems that support the *tirdwr* STREAMS module, if that module is pushed onto a stream corresponding to a transport provider supporting the TPI, then the **write(2)** operating system service routine may be used to send data on that transport endpoint. In this case there are no implied transport service data unit boundaries. Data is passed down the stream as a series of M_DATA messages.

This primitive does not require any acknowledgments, although it may generate a fatal error. This is indicated via a M_ERROR message type which results in the failure of all operating system service routines on the stream.

ERRORS

The allowable errors are as follows:

[EPROTO]

This indicates one of the following unrecoverable protocol conditions:

- The transport service interface was found to be in an incorrect state. If the interface is in the TS_IDLE state when the provider receives the T_DATA_REQ primitive, then the transport provider should just drop the message without generating a fatal error.
- The amount of transport user data associated with the primitive defines a transport service data unit larger than that allowed by the transport provider.

MODES

Only connection-mode.

ORIGINATOR

Transport user.

NAME

T_DISCON_IND - Disconnect Indication

SYNOPSIS

This message consists of a M_PROTO message block formatted as follows:

```
struct T_discon_ind {
    t_scalar_t   PRIM_type;      /* Always T_DISCON_IND */
    t_scalar_t   DISCON_reason;
    t_scalar_t   SEQ_number;
};
```

DESCRIPTION

This primitive indicates to the user that either a request for connection has been denied or an existing connection has been disconnected. The format of this message is one M_PROTO message block possibly followed by one or more M_DATA message blocks if there is any user data associated with the primitive.

PARAMETERS

PRIM_type
identifies the primitive type

DISCON_reason
is the reason for disconnect. The reason codes are protocol specific.

SEQ_number
is the sequence number which identifies which connect indication was denied, or it is -1 if the provider is disconnecting an existing connection.

RULES

The SEQ_number is only meaningful when this primitive is sent to a passive user who has the corresponding connect indication outstanding. It allows the transport user to identify which of its outstanding connect indications is associated with the disconnect.

MODES

Only connection-mode.

ORIGINATOR

Transport provider.

NAME

T_DISCON_REQ - Disconnect Request

SYNOPSIS

This message consists of one M_PROTO message block followed by one or more M_DATA message blocks if there is any user data specified by the transport user. The format of the M_PROTO message block is as follows:

```
struct T_discon_req {
    t_scalar_t   PRIM_type;      /* Always T_DISCON_REQ */
    t_scalar_t   SEQ_number;
};
```

DESCRIPTION

This primitive requests that the transport provider deny a request for connection, or disconnect an existing connection.

PARAMETERS

PRIM_type
identifies the primitive type.

SEQ_number
identifies the outstanding connect indication that is to be denied. If the disconnect request is disconnecting an already existing connection, then the value of *SEQ_number* will be ignored.

RULES

This primitive requires the transport provider to generate the following acknowledgment on receipt of the primitive, and the transport user must wait for the acknowledgment before issuing any other primitives:

Successful
Correct acknowledgment of the primitive is indicated via the T_OK_ACK primitive described in reference **TPI-SMD**.

Non-fatal errors
These errors will be indicated via the T_ERROR_ACK primitive described in reference **TPI-SMD**.

ERRORS

The allowable errors are as follows:

[TBADDATA]
The amount of user data specified was invalid.

[TBADSEQ]
The sequence number specified in the primitive was incorrect or invalid.

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

[TOUTSTATE]
The primitive would place the transport interface out of state.

[TSYSERR]
A system error has occurred and the UNIX system error is indicated in the primitive.

MODES

Only connection-mode.

ORIGINATOR

Transport user.

NAME

T_ERROR_ACK - Error Acknowledgment

SYNOPSIS

This message consists of a M_PCPROTO message block formatted as follows:

```
struct T_error_ack {
    t_scalar_t   PRIM_type;      /* Always T_ERROR_ACK */
    t_scalar_t   ERROR_prim;     /* Primitive in error */
    t_scalar_t   TLI_error;
    t_scalar_t   UNIX_error;
};
```

DESCRIPTION

This primitive indicates to the transport user that a non-fatal error has occurred in the last transport-user-originated primitive.

For an overview of the error handling capabilities available to the transport provider see reference **TPI-SMD**.

PARAMETERS

PRIM_type
identifies the primitive.

ERROR_prim
identifies the primitive type that caused the error

TLI_error
contains the Transport Level Interface error code.

UNIX_error
contains the UNIX system error code. This may only be non zero if TLI_error is equal to TSYSERR.

RULES

This may only be initiated as an acknowledgment for those primitives that require one. It also indicates to the user that no action was taken on the primitive that caused the error.

ERRORS

The list of Transport Level Interface error codes are listed in Appendix F of the referenced **XNS** specification.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport provider.

NAME

T_EXDATA_IND - Expedited Data Indication

SYNOPSIS

This message consists of one M_PROTO message block followed by one or more M_DATA message blocks containing at least one byte of data. The format of the M_PROTO message block is as follows:

```
struct T_exdata_ind {
    t_scalar_t   PRIM_type;      /* Always T_EXDATA_IND */
    t_scalar_t   MORE_flag;
};
```

DESCRIPTION

This primitive indicates to the transport user that this message contains an expedited transport interface data unit. One or more expedited transport interface data units form an expedited transport service data unit.

This primitive has a mechanism which indicates the beginning and end of an expedited transport service data unit. However, not all transport providers support the concept of an expedited transport service data unit.

PARAMETERS

PRIM_type
identifies the primitive type.

MORE_flag
when greater than zero, indicates that the next T_EXDATA_IND primitive is also part of this expedited transport service data unit.

MODES

Only connection-mode.

ORIGINATOR

Transport provider.

NAME

T_EXDATA_REQ - Expedited Data Request

SYNOPSIS

This message consists of one M_PROTO message block followed by one or more M_DATA message blocks containing at least one byte of data. The format of the M_PROTO message block is as follows:

```
struct T_exdata_req {
    t_scalar_t   PRIM_type;      /* Always T_EXDATA_REQ */
    t_scalar_t   MORE_flag;
};
```

DESCRIPTION

This primitive indicates to the transport provider that this message contains an expedited transport interface data unit. One or more expedited transport interface data units form an expedited transport service data unit.

Note that the maximum size of a expedited transport service data unit is indicated to the transport user via the T_INFO_ACK primitive.

This primitive has a mechanism which indicates the beginning and end of an expedited transport service data unit. However, not all transport providers support the concept of an expedited transport service data unit.

PARAMETERS

PRIM_type
: identifies the primitive type.

MORE_flag
: when greater than zero indicates that the next T_EXDATA_REQ primitive is also part of this expedited transport service data unit.

RULES

This primitive does not require any acknowledgments, although it may generate a fatal error. This is indicated via a M_ERROR message type which results in the failure of all operating system service routines on the stream.

ERRORS

The allowable errors are as follows:

[EPROTO]
: This indicates one of the following unrecoverable protocol conditions:

- The transport service interface was found to be in an incorrect state. If the interface is in the TS_IDLE state when the provider receives the T_EXDATA_REQ primitive, then the transport provider should just drop the message without generating a fatal error.
- The amount of transport user data associated with the primitive defines an expedited transport service data unit larger than that allowed by the transport provider.

MODES

Only connection-mode.

ORIGINATOR

Transport user.

NAME

T_INFO_ACK - Protocol Information Acknowledgment

SYNOPSIS

This message consists of a M_PCPROTO message block formatted as follows:

```
struct T_info_ack {
    t_scalar_t   PRIM_type;      /* Always T_INFO_ACK    */
    t_scalar_t   TSDU_size;      /* Max TSDU size        */
    t_scalar_t   ETSDU_size;     /* Max ETSDU size       */
    t_scalar_t   CDATA_size;     /* Connect data size    */
    t_scalar_t   DDATA_size;     /* Disconnect data size */
    t_scalar_t   ADDR_size;      /* TSAP size            */
    t_scalar_t   OPT_size;       /* Options size         */
    t_scalar_t   TIDU_size;      /* TIDU size            */
    t_scalar_t   SERV_type;      /* Service type         */
    t_scalar_t   CURRENT_state;  /* Current state        */
    t_scalar_t   PROVIDER_flag;  /* Provider flag        */
};
```

DESCRIPTION

This primitive indicates to the transport user any relevant protocol-dependent parameters. It should be initiated in response to the T_INFO_REQ primitive described above. The format of this message is one M_PCPROTO message block.

PARAMETERS

The fields of this message have the following meanings:

PRIM_type

This indicates the primitive type.

TSDU_size

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

ETSDU_size

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.

CDATA_size

A value greater than or equal to zero specifies the maximum amount of data that may be associated with connection establishment primitives; and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment primitives.

DDATA_size

A value greater than or equal to zero specifies the maximum amount of data that may be associated with the disconnect primitives; and a value of -2 specifies that the transport provider does not allow data to be sent with the disconnect primitives.

ADDR_size
A value greater than or equal to 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.

OPT_size
A value greater than or equal to 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.

TIDU_size
This is the amount of user data that may be present in a single T_DATA_REQ or T_EXDATA_REQ primitive. This is the size of the transport protocol interface data unit, and should not exceed the tunable system limit, if non-zero, for the size of a STREAMS message.

SERV_type
This field specifies the service type supported by the transport provider, and is one of the following:

T_COTS
The provider service is connection oriented with no orderly release support.

T_COTS_ORD
The provider service is connection oriented with orderly release support.

T_CLTS
The provider service is a connectionless transport service.

CURRENT_state
This is the current state of the transport provider.

PROVIDER_flag
This field specifies additional properties specific to the transport provider and may alter the way the transport user communicates. The following flags may be set by the provider:

SENDZERO
This flag indicates that the transport provider supports the sending of zero-length TSDUs.

XPG4_1
This flag indicates that the transport provider supports XPG4 semantics.

RULES

The following rules apply when the type of service is T_CLTS:

- The ETSDU_size, CDATA_size and DDATA_size fields should be -2.
- The TSDU_size should equal the TIDU_size.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport provider.

NAME

T_INFO_REQ - Get Transport Protocol Parameter Sizes

SYNOPSIS

This message consists of a M_PCPROTO message block formatted as follows:

```
struct T_info_req {
    t_scalar_t   PRIM_type;      /* Always T_INFO_REQ */
};
```

DESCRIPTION

This primitive requests the transport provider to return the sizes of all relevant protocol parameters, plus the current state of the provider.

PARAMETERS

PRIM_type
indicates the primitive type.

Note that the T_INFO_REQ and T_INFO_ACK primitives have no effect on the state of the transport provider and do not appear in the state tables.

RULES

This primitive requires the transport provider to generate one of the following acknowledgments on receipt of the primitive and that the transport user wait for the acknowledgment before issuing any other primitives:

Successful
Acknowledgment of the primitive via the T_INFO_ACK.

Non-fatal errors
There are no errors associated with this primitive.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport user.

NAME

T_OK_ACK - Success Acknowledgment

SYNOPSIS

This message consists of one M_PCPROTO message block formatted as follows:

```
struct T_ok_ack {
    t_scalar_t   PRIM_type;      /* Always T_OK_ACK */
    t_scalar_t   CORRECT_prim;
};
```

DESCRIPTION

This primitive indicates to the transport user that the previous transport-user-originated primitive was received successfully by the transport provider. It does not indicate to the transport user any transport protocol action taken due to issuing the T_INFO_REQ primitive. This may only be initiated as an acknowledgment for those primitives that require one.

PARAMETERS

PRIM_type
identifies the primitive.

CORRECT_prim
contains the successfully received primitive type.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport provider.

NAME

T_OPTMGMT_ACK - Option Management Acknowledgment

SYNOPSIS

This message consists of a M_PCPROTO message block formatted as follows:

```
struct T_optmgmt_ack {
    t_scalar_t   PRIM_type;      /* Always T_OPTMGMT_ACK */
    t_scalar_t   OPT_length;
    t_scalar_t   OPT_offset;
    t_scalar_t   MGMT_flags;
};
```

DESCRIPTION

This indicates to the transport user that the options management request has completed.

PARAMETERS

PRIM_type
: indicates the primitive type

OPT_length
: is the length of the protocol options associated with the primitive

OPT_offset
: is the offset from the beginning of the M_PCPROTO block where the options begin.

The proper alignment of the options is not guaranteed. *MGMT_flags* should be the same as those specified in the T_OPTMGMT_REQ primitive with any additional flags as specified below.

RULES

The following rules apply to the T_OPTMGMT_ACK primitive.

- If the value of *MGMT_flags* in the T_OPTMGMT_REQ primitive is T_DEFAULT, the provider should return the default provider options without changing the existing options associated with the stream.
- If the value of *MGMT_flags* in the T_OPTMGMT_REQ primitive is T_CHECK, the provider should return the options as specified in the T_OPTMGMT_REQ primitive along with the additional flags T_SUCCESS or T_FAILURE which indicate to the user whether the specified options are supportable by the provider. The provider should not change any existing options associated with the stream.
- If the value of *MGMT_flags* in the T_OPTMGMT_REQ primitive is T_NEGOTIATE , the provider should set and negotiate the option as specified by the following rules:
  - If the *OPT_length* field of the T_OPTMGMT_REQ primitive is 0, then the transport provider is to set and return the default options associated with the stream in the T_OPTMGMT_ACK primitive.
  - If options are specified in the T_OPTMGMT_REQ primitive, then the transport provider should negotiate those options, set the negotiated options and return the negotiated options in the T_OPTMGMT_ACK primitive. It is the user's responsibility to check the negotiated options returned in the T_OPTMGMT_ACK primitive and take appropriate action.
- If the value of *MGMT_flags* in the T_OPTMGMT_REQ primitive is T_CURRENT, the provider should return the currently effective option values without changing any existing options associated with the stream.

ERRORS

If the above rules result in an error condition, the transport provider must issue a T_ERROR_ACK primitive to the transport user specifying the error as defined in the description of the T_OPTMGMT_REQ primitive.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport provider.

NAME

T_OPTDATA_REQ - Data request with options

SYNOPSIS

The message consists of one M_PROTO message block followed by zero or more message blocks, where each M_DATA message block contains zero or more bytes of data. The format of the M_PROTO message block is as follows:

```
struct T_optdata_req {
    t_scalar_t PRIM_type;     /* always T_OPTDATA_REQ           */
    t_scalar_t DATA_flag;     /* flag bits associated with data */
    t_scalar_t OPT_length;    /* options length                 */
    t_scalar_t OPT_offset;    /* options offset                 */
};
```

DESCRIPTION

The primitive indicates to the transport provider that the message contains a transport interface data unit. One or more transport interface data units form a transport service data units (TSDU).

Note that the maximum transport service and data unit sizes allowed by transport provider is indicated to the user by T_INFO_ACK primitive.

This primitive has a mechanism that indicates the beginning and end of a transport service data unit. However not all transport providers support the concept of a transport service data unit.

This primitive also provides mechanisms to have options associated with the data being transferred.

PARAMETERS

The fields of this message have the following semantics:

PRIM_type
identifies the primitive type

DATA_flag
This field specifies bit fields specific general properties associated with the data being transferred. The following settings are currently defined:

T_ODF_MORE	When set, this bit indicates that the next T_OPTDATA_REQ primitive is also part of this transport service data unit.

OPT_length
the length of the requested options asociated with the primitive

OPT_offset
the offset (from the beginning of the M_PROTO message block) where the options asociated with this primitive begin.

RULES

It is possible to use this primitive with no associated options, in which case the *OPT_length* field is zero.

The primitive does not require any acknowledgements, although it may generate a fatal error. This is indicated via a M_ERROR message type, which results in the failure of all operating system service routines on the stream.

ERRORS

The allowable errors are as follows:

[EPROTO]

This indicates of the following unrecoverable protocol conditions:

- The transport service interface was found to be in an incorrect state. If the interface is in TS_IDLE state when the provider receives the T_OPTDATA_REQ primitive, then the transport provider should just drop the message without generating a fatal error.
- The amount of transport user data associated with the primitive defines a transport service data unit larger than that allowed by the transport provider.

MODES

Only connection mode.

ORIGINATOR

Transport provider.

NAME

T_OPTDATA_REQ - Data request with options

SYNOPSIS

The message consists of one M_PROTO message block followed by zero or more message blocks, where each M_DATA message block contains zero or more bytes of data. The format of the M_PROTO message block is as follows:

```
struct T_optdata_req {
   t_scalar_t PRIM_type;       /* always T_OPTDATA_REQ           */
   t_scalar_t DATA_flag;       /* flag bits associated with data */
   t_scalar_t OPT_length;      /* options length                 */
   t_scalar_t OPT_offset;      /* options offset                 */
};
```

DESCRIPTION:

The primitive indicates to the transport provider that the message contains a transport interface data unit. One or more transport interface data units form a transport service data units (TSDU).

Note that the maximum transport service and data unit sizes allowed by transport provider is indicated to the user by the T_INFO_ACK primitive.

This primitive has a mechanism that indicates the beginning and end of a transport service data unit. However not all transport providers support the concept of a transport service data unit.

This primitive also provides mechanisms to have options associated with the data being transferred.

PARAMETERS

The fields of this message have the following semantics:

PRIM_type

identifies the primitive type

DATA_flag

This field specifies bit fields specific general properties associated with the data being transferred. The following settings are currently defined:

T_ODF_MORE — When set, this bit indicates that the next T_OPTDATA_REQ primitive is also part of this transport service data unit.

OPT_length

the length of the requested options asociated with the primitive

OPT_offset

the offset (from the beginning of the M_PROTO message block) where the options asociated with this primitive begin.

RULES

It is possible to use this primitive with no associated options, in which case the *OPT_length* field is zero.

The primitive does not require any acknowledgements, although it may generate a fatal error. This is indicated via a M_ERROR message type, which results in the failure of all operating system service routines on the stream.

ERRORS

The allowable errors are as follows:

[EPROTO]

This indicates of the following unrecoverable protocol conditions:

- The transport service interface was found to be in an incorrect state. If the interface is in TS_IDLE state when the provider receives the T_OPTDATA_REQ primitive, then the transport provider should just drop the message without generating a fatal error.
- The amount of transport user data associated with the primitive defines a transport service data unit larger than that allowed by the transport provider.

MODES

Only connection mode

ORIGINATOR

Transport user

NAME

T_OPTMGMT_REQ - Options Management

SYNOPSIS

This message consists of a M_PROTO message block formatted as follows:

```
struct T_optmgmt_req {
    t_scalar_t   PRIM_type;      /* Always T_OPTMGMT_REQ */
    t_scalar_t   OPT_length;
    t_scalar_t   OPT_offset;
    t_scalar_t   MGMT_flags;
};
```

DESCRIPTION

This primitive allows the transport user to manage the options associated with the stream. The format of the message is one M_PROTO message block.

PARAMETERS

PRIM_type
indicates the primitive type

OPT_length
is the length of the protocol options associated with the primitive

OPT_offset
is the offset from the beginning of the M_PROTO block where the options begin.

MGMT_flags
are the flags which define the request made by the transport user. The allowable flags are:

T_NEGOTIATE
Negotiate and set the options with the transport provider

T_CHECK
Check the validity of the specified options

T_DEFAULT
Return the default options

T_CURRENT
Return the currently effective option values.

The proper alignment of the options is not guaranteed. The options are, however, aligned the same as received from the transport user.

RULES

For the rules governing the requests made by this primitive see the T_OPTMGMT_ACK primitive.

This primitive requires the transport provider to generate one of the following acknowledgments on receipt of the primitive, and that the transport user wait for the acknowledgment before issuing any other primitives:

Successful
Acknowledgment of the primitive via the T_OPTMGMT_ACK.

Non-fatal errors
These errors will be indicated via the T_ERROR_ACK primitive described in Section 1.3 on page 3.

ERRORS

The allowable errors are as follows:

TACCES
: The user did not have proper permissions for the use of the requested options.

TBADFLAG
: The flags as specified were incorrect or invalid.

TBADOPT
: The options as specified were in an incorrect format, or they contained invalid information.

TOUTSTATE
: The primitive would place the transport interface out of state.

TNOTSUPPORT
: This primitive is not supported by the transport provider.

TSYSERR
: A system error has occurred and the UNIX system error is indicated in the primitive.

MODES

Both connection-mode and connectionless-mode.

Originator

Transport user.

NAME

T_ORDREL_IND - Orderly Release Indication

SYNOPSIS

This message consists of a M_PROTO message block formatted as follows:

```
struct T_ordrel_ind {
    t_scalar_t   PRIM_type;      /* Always T_ORDREL_IND */
};
```

DESCRIPTION

This primitive indicates to the transport user that the user on the other side of the connection is finished sending data. This primitive is only supported by the transport provider if it is of type T_COTS_ORD.

PARAMETERS

PRIM_type
identifies the primitive type.

MODES

Only connection-mode.

ORIGINATOR

Transport provider.

NAME

T_ORDREL_REQ - Orderly Release Request

SYNOPSIS

This message consists of a M_PROTO message block formatted as follows:

```
struct T_ordrel_req {
    t_scalar_t   PRIM_type;      /* Always T_ORDREL_REQ */
};
```

DESCRIPTION

This primitive indicates to the transport provider that the user is finished sending data. This primitive is only supported by the transport provider if it is of type T_COTS_ORD.

PARAMETERS

PRIM_type
identifies the primitive type.

RULES

This primitive does not require any acknowledgments, although it may generate a fatal error. This is indicated via a M_ERROR message type which results in the failure of all operating system service routines on the stream.

ERRORS

[EPROTO]
This indicates the unrecoverable protocol condition that the primitive would place the interface in an incorrect state.

MODES

Only connection-mode.

ORIGINATOR

Transport user.

NAME

T_UDERROR_IND **-** Unitdata Error Indication

SYNOPSIS

This message consists of a M_PROTO message block formatted as follows:

```
struct T_uderror_ind {
    t_scalar_t   PRIM_type;      /* Always T_UDERROR_IND */
    t_scalar_t   DEST_length;
    t_scalar_t   DEST_offset;
    t_scalar_t   OPT_length;
    t_scalar_t   OPT_offset;
    t_scalar_t   ERROR_type;
};
```

DESCRIPTION

This primitive indicates to the transport user that a datagram with the specified destination address and options produced an error.

PARAMETERS

PRIM_type
: identifies the primitive type.

DEST_length
: is the length of the destination address.

DEST_offset
: is the offset (from the beginning of the M_PROTO message block) where the destination address begins.

OPT_length
: is the length of the requested options associated with the primitive.

OPT_offset
: is the offset (from the beginning of the M_PROTO message block) of the requested options associated with the primitive.

ERROR_type
: defines the protocol dependent error code.

The proper alignment of the destination address and options in the M_PROTO message block is not guaranteed.

MODES

Only connectionless-mode.

ORIGINATOR

Transport provider.

NAME

T_UNBIND_REQ - Unbind Protocol Address Request

SYNOPSIS

This message consists of a M_PROTO message block formatted as follows:

```
struct T_unbind_req {
    t_scalar_t   PRIM_type;      /* Always T_UNBIND_REQ */
};
```

DESCRIPTION

This primitive requests that the transport provider unbind the protocol address associated with the stream and deactivate the stream.

PARAMETERS

PRIM_type
indicates the primitive type.

RULES

This primitive requires the transport provider to generate the following acknowledgments on receipt of the primitive and that the transport user must wait for the acknowledgment before issuing any other primitives:

Successful
Correct acknowledgment of the primitive is indicated via the T_OK_ACK primitive described in reference **TPI-SMD**.

Non-fatal errors
These errors will be indicated via the T_ERROR_ACK primitive described in reference **TPI-SMD**.

ERRORS

The allowable errors are as follows:

[TOUTSTATE]
The primitive would place the transport interface out of state.

[TSYSERR]
A system error has occurred and the UNIX System error is indicated in the primitive.

MODES

Both connection-mode and connectionless-mode.

ORIGINATOR

Transport user.

NAME

T_UNITDATA_IND - Unitdata Indication

SYNOPSIS

This message consists of one M_PROTO message block followed by zero or more M_DATA message blocks where each M_DATA message block contains at least one byte of data. The format of the M_PROTO message block is as follows:

```
struct T_unitdata_ind {
    t_scalar_t   PRIM_type;      /* Always T_UNITDATA_IND */
    t_scalar_t   SRC_length;
    t_scalar_t   SRC_offset;
    t_scalar_t   OPT_length;
    t_scalar_t   OPT_offset;
};
```

DESCRIPTION

This primitive indicates to the transport user that a datagram has been received from the specified source address.

PARAMETERS

PRIM_type
identifies the primitive type.

SRC_length
is the length of the source address.

SRC_offset
is the offset (from the beginning of the M_PROTO message block) where the source address begins.

OPT_length
is the length of the requested options associated with the primitive.

OPT_offset
is the offset (from the beginning of the M_PROTO message block) of the requested options associated with the primitive.

The proper alignment of the source address and options in the M_PROTO message block is not guaranteed.

MODES

Only connectionless-mode.

ORIGINATOR

Transport provider.

NAME

T_UNITDATA_REQ - Unitdata Request

SYNOPSIS

This message consists of one M_PROTO message block followed by zero or more M_DATA message blocks where each M_DATA message block contains zero or more bytes of data. The format of the M_PROTO message block is as follows:

```
struct T_unitdata_req {
    t_scalar_t   PRIM_type;      /* Always T_UNITDATA_REQ */
    t_scalar_t   DEST_length;
    t_scalar_t   DEST_offset;
    t_scalar_t   OPT_length;
    t_scalar_t   OPT_offset;
};
```

DESCRIPTION

This primitive requests that the transport provider send the specified datagram to the specified destination.

PARAMETERS

PRIM_type
identifies the primitive type.

DEST_length
is the length of the destination address

DEST_offset
is the offset (from the beginning of the M_PROTO message block) where the destination address begins.

OPT_length
is the length of the requested options associated with the primitive.

OPT_offset
is the offset (from the beginning of the M_PROTO message block) of the requested options associated with the primitive.

The proper alignment of the destination address and options in the M_PROTO message block is not guaranteed. The destination address and options in the M_PROTO message block are, however, aligned the same as they were received from the transport user.

This primitive does not require any acknowledgment. If a non-fatal error occurs, it is the responsibility of the transport provider to report it via the T_UDERROR_IND indication. Fatal errors are indicated via a M_ERROR message type which results in the failure of all operating system service routines on the stream.

ERRORS

The allowable fatal errors are as follows:

[EPROTO]
This indicates one of the following unrecoverable protocol conditions:

- The transport service interface was found to be in an incorrect state.
- The amount of transport user data associated with the primitive defines an transport service data unit larger than that allowed by the transport provider.

MODES

Only connectionless-mode.

ORIGINATOR

Transport user.

Connection Acceptance

Connection acceptance with TPI is not easy to understand without the benefit of knowing how it has evolved. This Appendix therefore offers background information to explain the state of affairs under existing common implementations, and hence assist the reader in understanding an existing implementation or designing a new one.

The following text is provided for informational purposes only and should not be construed as imposing normative requirements.

For brevity in the following discussion:

user	means a *transport user*
provider	means a *transport provider*
address	means a *transport address*
endpoint	means a *transport endpoint*.

A.1 Accepting Incoming Connections

In order to field an incoming connection request, a user must establish an endpoint and use the T_BIND_REQ message (with a CONIND_number greater than zero) to bind to the local address. The CONIND_number in that message expresses the number of outstanding incoming connection requests the endpoint should support. There may be more than one endpoint bound to the same local address, but only one of them at a time may have a CONIND_number greater than zero. Such an endpoint, if it exists, is called a *listener*. The other endpoints, if any, bound to the same address will either be conducting outgoing connections or carrying out incoming connections which were processed by a listener. There can only be one listener for each local address because the provider needs to know where to send any T_CONN_IND messages for that address.

Each listening endpoint can only be listening on one local address. When an incoming connection request is detected by the provider it looks for a matching listener in the TS_BIND state. If it does not find one, it fails the connection request, otherwise it constructs a T_CONN_IND message and sends it up the listener to the user. The user sends a T_CONN_RES if it wants to accept the connection, or a T_DISCON_REQ if it does not.

It is permissible for the listener to conduct the actual connection, but this is unusual in practice because, while it is doing so, it cannot also perform its listening task because it will be in some other state than TS_BIND. By far the more usual methodology is for the user to establish a new endpoint and use that for conducting the actual connection while the listener continues to listen for further incoming connections. The T_CONN_RES message contains a field *ACCEPTOR_id* which is used to identify the endpoint on which the user wishes to conduct the connection. The encoding of this field is implementation specific as are the methods of acquiring a valid value for it, and the method employed by the provider in interpreting it.

In older versions of the TPI standard the *ACCEPTOR_id* field was called *QUEUE_ptr* and had the type **queue_t ***. This unfortunately exposed an implementation detail which made the use of TPI difficult on systems where a pointer is a different length at different times (for example a 64-bit system supporting both 32-bit and 64-bit user applications), and also on systems where transport provider operates in a different address space from other parts of the operating system. Nevertheless, on many systems, the *ACCEPTOR_id* is still given the value of the provider queue pair read pointer of the endpoint which is to be used to conduct the connection. This remains a perfectly good implementation strategy for those systems which do not suffer the problems mentioned above. The value of the *QUEUE_ptr* variable was never used by the user as any more than an opaque identifier value (in fact most implementations did not even expose the value to the user).

A.2 The Common Single Type Model Implementation

The provider constructs a T_CONN_IND message with the source address of the originating (usually remote) user. It includes any (protocol specific) options and creates a unique reference number which it places in *SEQ_number*. The encoding and origin of this field is implementation specific under the constraint that it must be unique during the lifetime of the connection acceptance. Some implementations use the address of a kernel data structure associated with the connection request. Others use an incrementing counter and trust that less than 4,294,967,296 incoming connection requests do not occur on this provider before the user responds (this is a fairly safe assumption). The user is then sent the message on the listener.

When the user receives the T_CONN_IND message, it usually opens an entirely new endpoint (to the same transport provider). It may choose to bind that new endpoint to a local address, or it may leave the provider to perform that task on receipt of the T_CONN_RES. Any address it binds to must satisfy the requirements of the provider for the connection. The new endpoint should not have a *CONIND_number* greater than 0.

The user constructs a T_CONN_RES message. It copies in the *SEQ_number* from the T_CONN_IND (otherwise the transport will not know to which connection it is responding), removes (if necessary) the options it is not prepared to support, and copies the remainder into the T_CONN_RES . The T_CONN_RES is now complete except for the *ACCEPTOR_id*. The user does not directly have the information to include in this field; only the operating system kernel can derive that. The usual solution is for the kernel to supply a special **ioctl(2)** call called *I_FDINSERT* which expects as argument a T_CONN_RES message and the file-descriptor of the new (accepting) endpoint. This **ioctl(2)** call is specially treated. Before the message is sent down to the provider, the kernel uses the file-descriptor to access the endpoint. It extracts the value of the provider read queue pointer from that endpoint and places its value in the *ACCEPTOR_id* field. Then it sends the message to the provider.

The provider cross-references the *SEQ_number* and determines that it has such a pending connection, then it checks that the *ACCEPTOR_id* matches the read queue pointer of a valid endpoint (it must exist and obey all the general and provider specific rules). If it is not already bound to a local address the provider will bind it to the same address as that to which the listener is bound. If the *ACCEPTOR_id* identifies the listener, then the listener becomes the acceptor and further incoming connection requests for its address will fail, at least until the connection terminates. In the usual case, however, a new endpoint is used to conduct the new connection.

If the listener concocts an *ACCEPTOR_id* which does not represent one of its own endpoints, and gets it exactly correct, then it is possible that it could foist one of its own connections off onto an unsuspecting endpoint if it was in the correct state, etc. This could be a denial of service attack. What it cannot do, is to hijack a connection from another listener.

A.3 Possible Multiple Type Model Implementation Methodologies

On 64-bit systems, a decision needs to be made about how to provide a consistent *ACCEPTOR_id* which has the property of being unique within each transport provider. If the *I_FDINSERT* **ioctl(2)** call is still used, then the *ACCEPTOR_id* encoding must be based on data which is accessible to the STREAM head when the *I_FDINSERT* call is made. It is likely to be simplest just to encode the 64-bit value of the read queue pointer in the 32-bit *ACCEPTOR_id*, possibly simply by truncation. The key is to preserve the uniqueness of the value as an identifier.

The STREAM head generates the identifier and places the result in *ACCEPTOR_id*. When the T_CONN_RES message reaches the provider, it decodes the *ACCEPTOR_id* to identify the accepting endpoint.

Glossary

CLTS
Connectionless mode of service, in which the origin and destination addresses are included in each message packet so that a direct connection or established session between origin and destination is not required.

COTS
Connection-oriented mode of service, requiring a direct connection or established session between origin and destination.

IP
Internet Protocol

STREAMS
A feature of UNIX that provides a standard way of dynamically building and passing messages up and down a protocol stack. Upstream messages are passed from the network driver through the STREAMS modules to the application. Downstream messages flow from the application to the network driver. A STREAMS module would be a transport layer protocol (e.g. TCP) or a network layer protocol (e.g. IP).

TCP
Transmission Control Protocol

TPI
Transport Provider Interface

Type Model
A mapping of the C language fundamental types onto the data formats supported by a computer architecture. Examples of type models are ILP32 (char 8 bits, short 16 bits, int, long and pointer 32 bits), and LP64 (char 8 bits, short 16 bits, int 32 bits, long and pointer 64 bits).

UI
UNIX International. This organization developed the original specification for TPI. It was subsequently acquired by UNIX System Laboratories.

USL
UNIX System Laboratories. This organization acquired the TPI specification rights from UI. It was subsequently acquired by Novell Inc.

Index

32-bit ... 62
64-bit ... 63
address ... 61
allowable sequence of TPI primitives ... 9
CLTS ... 65
connection acceptance ... 61
connection-mode ... 2, 4
connection-oriented mode ... 14
connectionless mode ... 15
connectionless-mode ... 2, 4
COTS ... 65
data transfer state table for CLTS ... 15
data transfer state table for COTS ... 14
endpoint ... 61
EPROTO ... 4
event ... 9
fatal error ... 3
flushing queues ... 5
implementation-defined ... 19, 61
implementations ... 1
incoming events ... 12
initialization state table ... 13
IP ... 65
ISO IS 8072 ... 3
ISO IS 8072 / DAD ... 3
kernel level incoming events ... 12
kernel level outgoing events ... 11
kernel level states ... 10
message format ... 19
message interface ... 2
message type ... 7
multiple
  type ... **63**
M_DATA ... 2
M_ERROR ... 2, 4
M_FLUSH ... 2, 5
M_PCPROTO ... 2
M_PROTO ... 2
non-fatal error ... 3
OSI ... 1
outgoing events ... 11
precedence ... 17
  rules" ... 5"
  stream read queue ... 18
  stream write queue ... 17
primitive ... 2, 4, 9, 17
primitives list ... 7
protocol ... 1
provider ... 61
rules for flushing queues ... 5
rules for precedence ... 5
rules for TPI sequence of primitives ... 4
sequence of primitives ... 4
single type model ... 62
state ... 9
state table ... 10, 13
  data transfer for CLTS ... 15
  data transfer for COTS ... 14
  initialization ... 13
state table variables ... 11
stream read queue ... 18
stream write queue ... 17
streams ... 1-2
STREAMS ... 65
streams message ... 2
streams message type ... 7
TCP ... 65
TPI ... 65
TPI message formats ... 19
transport address ... 61
transport endpoint
  61
transport layer ... 1
transport primitive ... 2
transport primitive precedence ... 17
transport primitives ... 7
transport provider ... 2, 61
transport service definitions ... 3
transport service state table ... 13
transport user ... 9, 61
Type Model ... 65
T_ADDR_ACK ... **20**
T_ADDR_REQ ... **21**
T_BIND_ACK ... **22**
T_BIND_REQ ... **24**
T_CONN_CON ... **26**
T_CONN_IND ... **27**
T_CONN_REQ ... **28**
T_CONN_RES ... **30**
T_DATA_IND ... **33**
T_DATA_REQ ... **34**
T_DISCON_IND ... **36**

T_DISCON_REQ **37**
T_ERROR_ACK **39**
T_EXDATA_IND..**40**
T_EXDATA_REQ **41**
T_INFO_ACK **42**
T_INFO_REQ **44**
T_OK_ACK **45**
T_OPTDATA_IND **48**
T_OPTDATA_REQ **50**
T_OPTMGMT_ACK **46**
T_OPTMGMT_REQ **52**
T_ORDREL_IND **54**
T_ORDREL_REQ..**55**
T_UDERROR_IND..**56**
T_UNBIND_REQ..**57**
T_UNITDATA_IND **58**
T_UNITDATA_REQ..**59**
UI..65
user 61
USL 65
variables..11