
Unicenter

TCPaccess Communications Server Assembler API Concepts

Version 6.0



Computer Associates
The Software That Manages eBusiness



This documentation and related computer software program (hereinafter referred to as the "Documentation") is for the end user's informational purposes only and is subject to change or withdrawal by Computer Associates International, Inc. ("CA") at any time.

This documentation may not be copied, transferred, reproduced, disclosed or duplicated, in whole or in part, without the prior written consent of CA. This documentation is proprietary information of CA and protected by the copyright laws of the United States and international treaties.

Notwithstanding the foregoing, licensed users may print a reasonable number of copies of this documentation for their own internal use, provided that all CA copyright notices and legends are affixed to each reproduced copy. Only authorized employees, consultants, or agents of the user who are bound by the confidentiality provisions of the license for the software are permitted to have access to such copies.

This right to print copies is limited to the period during which the license for the product remains in full force and effect. Should the license terminate for any reason, it shall be the user's responsibility to return to CA the reproduced copies or to certify to CA that same have been destroyed.

To the extent permitted by applicable law, CA provides this documentation "as is" without warranty of any kind, including without limitation, any implied warranties of merchantability, fitness for a particular purpose or noninfringement. In no event will CA be liable to the end user or any third party for any loss or damage, direct or indirect, from the use of this documentation, including without limitation, lost profits, business interruption, goodwill, or lost data, even if CA is expressly advised of such loss or damage.

The use of any product referenced in this documentation and this documentation is governed by the end user's applicable license agreement.

The manufacturer of this documentation is Computer Associates International, Inc.

Provided with "Restricted Rights" as set forth in 48 C.F.R. Section 12.212, 48 C.F.R. Sections 52.227-19(c)(1) and (2) or DFARS Section 252.227-7013(c)(1)(ii) or applicable successor provisions.

© 2002 Computer Associates International, Inc. (CA)

All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Contents

Chapter 1: API Overview

References	1-2
Transport Service Modes	1-3
Background	1-4
ARPANET	1-4
TCP/IP	1-4
Corporate Networking Technology	1-5
OSI Reference Model.....	1-5
Open Systems Interconnection	1-5
The Seven Layer Model	1-6
Layer Interaction	1-6
Internet Protocol Suite	1-9
Major Application Layer Protocols.....	1-9
OSI Terminology.....	1-11
Transport Services and Protocols.....	1-11
Service Access Points (SAP).....	1-11
Connections.....	1-12
Primitives.....	1-12
Service Data Units	1-12
Transport Layer Services	1-13
Modes of Service.....	1-14
Connection-Oriented Transport Service (COTS)	1-14
Transport Connection Establishment	1-16
Data Transfer.....	1-18
Transport Connection Release.....	1-21
Endpoint States and Service Sequence	1-22
Connectionless Transport Service	1-24
Transport Layer Addressing.....	1-26
Internet Domain Addressing.....	1-26
Connecting to an Internet Address.....	1-29

Chapter 2: Concepts and Facilities

API Organization.....	2-2
Relationship Independence.....	2-2
Service Request Processing.....	2-3
API Components.....	2-4
Concepts and Terminology.....	2-5
Modes of Service.....	2-5
Endpoints and Access Points.....	2-6
Connection Strategies.....	2-7
Data Transfer Modes.....	2-10
Disconnect and Orderly Release.....	2-11
Service Requests and Parameters.....	2-11
Transport Protocol Options.....	2-17
The Transport Service Parameter List.....	2-17
Establishing a Session with a Transport Provider.....	2-23
Session-Level Services.....	2-23
Application Programs and Transport Users.....	2-25
Connection-Mode Service.....	2-27
Local Endpoint Management.....	2-28
Opening and Closing Endpoints.....	2-29
Opening an Endpoint.....	2-30
Closing an Endpoint.....	2-32
Passing Control of an Endpoint.....	2-33
Binding and Unbinding Addresses.....	2-34
Retrieving Protocol Addresses.....	2-37
Miscellaneous Functions.....	2-38
TINFO – Getting Basic Protocol Information.....	2-38
TOPTION – Manipulating Protocol Options.....	2-39
TUSER – Specifying or Changing an Endpoint User ID.....	2-41
Connection Establishment.....	2-42
TCONNECT – Initiating a Connection.....	2-43
Single-Threaded and Multithreaded Servers.....	2-44
Single-Threaded Servers.....	2-44
Multithreaded Servers.....	2-46
Data Transfer.....	2-48
TLI vs. Sockets Mode.....	2-48
Connection-Oriented Transport Service (COTS) Data Transfer Functions.....	2-48
Transporting User Data.....	2-49
Sending and Receiving Data.....	2-51
User Data Length.....	2-53
Connection Release.....	2-53

TDISCONN – Initiating Abortive Release	2-54
TCLEAR – Return Disconnect Information	2-54
Using TDISCONN and TCLEAR During Connection Establishment	2-55
TRELEASE – Orderly Release Procedure	2-55
TRELACK – Checking for Orderly Release	2-56
Connectionless-Mode Service	2-57
Local Endpoint Management.....	2-57
TOPEN – Opening an Endpoint.....	2-58
Using TBIND with Connectionless Mode.....	2-59
Terminating a Connectionless Mode Endpoint	2-59
Using TADDR to Retrieve Addresses	2-59
Specifying Protocol Options	2-60
Data Transfer.....	2-60
Connectionless Service with Associations	2-63
ASSOC – Requesting Association-Mode Service	2-64
Establishing Client Associations	2-65
Establishing Server Associations.....	2-66
Local Endpoint Control	2-67
TCHECK	2-68
TERROR – Abnormally Completed Service Requests.....	2-69
TEXEC – Executing a Fully-initialized TPL	2-69
TSTATE – Return Endpoint State.....	2-70
Declarative Macro Instructions.....	2-70
The APCB Macro Instruction.....	2-71
The TDSECT Macro Instruction	2-71
The TEXTLST Macro Instruction	2-71
The TEVNTLST Macro Instruction	2-72
The TPL Macro Instruction	2-72
Endpoint States and Function Sequences.....	2-73
Endpoint Functions	2-73
Endpoint States	2-73
State Transitions	2-75
Function Sequences.....	2-79

Chapter 3: Program Synchronization and Control

Task Synchronization Requirements	3-2
Typical Processing Flow	3-2
Modes of Operation.....	3-5
Operating Mode Differences.....	3-6
Synchronous Operation.....	3-8

Asynchronous Operation	3-9
Asynchronous Operation Using ECBs	3-11
Mixing Synchronization Modes	3-17
Specifying and Using Exit Routines.....	3-18
How Exit Routines Are Specified.....	3-18
TPL Exit Routines	3-18
TEXTLST and TEVNTLST Exit Routines.....	3-19
How Exit Routines Are Called	3-20
Synchronous Exit Routines	3-20
Asynchronous Exit Routines	3-23
Exit Routine Parameter List.....	3-26
How Exit Routines Are Used.....	3-27
Exit Routine Summary.....	3-27
Register Usage Summary	3-29
TPL Completion Exit.....	3-30
Protocol Event Exits and ECBs	3-31
SYNAD/LERAD—Synchronous Error Recovery Exits.....	3-37
TPEND Exit Routine or ECB.....	3-42
APEND Exit Routine.....	3-44
Deriving Context in Exit Routines	3-46
Handling Errors and Special Conditions	3-47
Macro Information	3-47
General Return Codes.....	3-48
Conditional Completion Codes	3-49
Recovery Action Codes.....	3-50
Specific Error Codes	3-51
Diagnostic Codes	3-51
AOPEN and ACLOSE Errors.....	3-52
Application Program Organization	3-53
Multitasking Operation Rules.....	3-54
Multiple Address Spaces	3-55
24-Bit and 31-Bit Addressing	3-55

Appendix A: Endpoint State Transitions

Defined Endpoint States	A-1
The State Transition Tables	A-2
Endpoint States for TSCLOSED, TSOPENED, and TSDSABLED	A-3
Endpoint States for TSENABLD, TSINCONN, and TSOUCONN.....	A-4
Endpoint States for TSCONNECT, TSINRLSE, and TSOURLSE.....	A-6

Appendix B: Time-Sequence Diagrams

Diagram Labeling	B-1
Synchronous and Asynchronous Modes	B-2
Completion and Error Events	B-2
Diagrams	B-3
Local Endpoint Management (Initialization)	B-3
Client Connect Sequence (Rejected).....	B-5
Server Connect Sequence (Accepted)	B-6
Server Connect Sequence (Rejected)	B-7
COTS Receive Data Sequence	B-8
COTS Send Data Sequence.....	B-9
CLTS Receive Data Sequence.....	B-9
CLTS Send Data Sequence	B-10
CLTS Datagram Error Sequence.....	B-11
Orderly Release Sequence	B-12
Abortive Disconnect Sequence	B-14
Simultaneous Disconnects.....	B-15
Local Endpoint Management (Termination)	B-16
Protocol Address	C-1
Transport Layer Address – Port Numbers.....	C-2
Network Layer Addressing – IP Address	C-3
Expedited Data.....	C-4
Sending Expedited Data	C-4
Receiving Expedited Data	C-4
Disconnect Reason Codes.....	C-5
API-Initiated Protocol Actions.....	C-6
Protocol Events Resulting In API Activity	C-7
Initial SYN Arrives (TCP).....	C-7
SYN/ACK Arrives in Response to a Previously Sent Initial SYN(TCP)	C-7
Data Arrives (TCP/UDP).....	C-7
Acknowledgment for Sent Data Arrives (TCP).....	C-7
Urgent Data Arrives (TCP).....	C-7
ICMP Message Arrives	C-8
A TCP RESET Arrives.....	C-8
A FIN Arrives (TCP)	C-8
Data Sets.....	D-1
Sample Program	E-1

Index

API Overview

This chapter provides an introduction and overview to the Unicenter TCPAccess Communications Server Application Program Interface (API).

It includes these sections:

- [Background](#) – Describes the history of multi-vendor computer networks, including a description of ARPANET, internet, corporate networking technology, and international standards
- [OSI Reference Model](#) – Describes open systems interconnection concepts and the internet protocol suite
- [OSI Reference Model](#) – Defines common OSI terms and describes their use, including service access points, connections, primitives, and service data units
- [Transport Layer Services](#) – Describes the types of services available for transporting data between networks, as well as modes of service and transport layer addressing

The Unicenter TCPAccess API is a programming interface between application programs and communication subsystems based on open network protocols. The API lets any application program operating in its own MVS address space to access and use communication services provided by an MVS subsystem that implements this interface. Unicenter TCPAccess provides communication services using TCP/IP protocols, and is an example of such a subsystem.

This guide describes an interface to the transport layer of the Basic Reference Model of Open Systems Interconnection (OSI). Although the API is capable of interfacing to proprietary protocols, the internet open network protocols are the intended providers of the transport service. This document uses the term *open* to emphasize that any system conforming to one of these standards can communicate with any other system conforming to the same standard, regardless of vendor. These protocols are contrasted with proprietary protocols that generally support a closed community of systems supplied by a single vendor.

References

For information about the Basic C Library and Socket API, refer to the *Unicenter TCPaccess Communications Server C/Socket Programmers Guide*. The RPC interface to the API is described in the *Unicenter TCPaccess Communications Server RPC/XDR Programmer's Reference*.

Refer to the these documents for additional information:

- *TCP/IP – Architecture, Protocols, and Implementation*, Sidnie Feit, McGraw-Hill, 1993.
- *DDN Protocol Handbook, Volume 2: DARPA Internet Protocols*, DDN Network Information Center, SRI International, Menlo Park, California, 1985.
- *Handbook of Computer Communications Standards, Volume 1: The Open Systems Interconnection (OSI) Model and OSI-Related Standards*, William Stallings, Macmillan Publishing Company, New York, 1987.
- *Handbook of Computer Communications Standards, Volume 3: Department of Defense (DOD) Protocol Standards*, William Stallings, Macmillan Publishing Company, New York, 1987.
- *Information Processing Systems, Open Systems Interconnection, Basic Reference Model*, ISO-7498 and ISO-7498/Add.1, International Organization for Standardization, Switzerland, 1984.
- *Information Processing Systems, Open Systems Interconnection, Transport Service Definition*, ISO-8072 and ISO-8072/Add.1, International Organization for Standardization, Switzerland, 1986.
- *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, Douglas Comer, Prentice Hall, New Jersey, 1988.
- *The Open Book: A Practical Perspective on OSI*, Marshall T. Rose, Prentice Hall, 1989.
- *UNIX System V Release 3 Network Programmers Guide*, P/N 307-230, AT&T, 1986.
- *UNIX System V Release 3 Programmer's Reference Manual*, Section 3N, P/N 307-226, AT&T, 1986.

Transport Service Modes

This document defines connection-mode and connectionless-mode protocols that support two basic modes of transport service:

- Connection-mode Protocols

With connection-mode protocols, a virtual connection is established, over which an ordered, reliable stream of data bytes or messages can be transferred until the connection is released. Each connection is associated with a specific transport user.

- Connectionless-mode Protocols

Connectionless-mode protocols provide a stateless form of data transfer where each message or datagram is unrelated to previous or successive datagrams, and can be received from or sent to any transport user.

Unicenter TCPAccess API is compliant with ISO International Standard 8072 (CCITT X.214) and ISO 8072/ Add.1, which specify the transport service definition for connection-mode and connectionless-mode service. The semantics of the interface were derived from the Transport Layer Interface (TLI) that is used by AT&T UNIX System V.3 for communicating with networks based on the OSI Reference Model. The mechanics for synchronizing with other MVS tasks and handling exceptional conditions are derived from IBM's Virtual Telecommunications Access Method (VTAM) to simplify learning the rudiments of the interface. Therefore, knowledge of VTAM application programming is helpful.

Note: The basic interface is implemented for application programs written in assembler language, and a working knowledge of IBM Basic Assembler Language (BAL) is assumed.

A library of basic interface functions is also provided for applications written in C language. This library implements the C-language equivalent of the basic functions defined for assembler language. For application programs that are derived or ported from BSD UNIX, a library of functions based on BSD sockets is also provided. Subroutine libraries for other high-level languages will be developed as the need arises.

Background

This section provides a brief description of how multi-vendor computer networks began and how they evolved to what they are today.

ARPANET

Research into multi-vendor computer networks began in the mid 1960s, and the first experimental network was deployed in 1969. This effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and led to the development of the largest network of computer systems in the world today. The original network, known as ARPANET, is still in operation.

The original communication protocols used on ARPANET have since been replaced by more advanced protocols that not only permit larger and more sophisticated networks, but also permit separate networks with their own administration to be connected together into a super-network. This network of networks has come to be known as the Internet and currently consists of tens-of-thousands of computer systems connected to thousands of interconnected networks. Any computer system on any one of these individual networks has the capability to communicate with any other computer system on some other network halfway around the world.

TCP/IP

The set of communication protocols that evolved out of this experimental effort is known as the Internet (TCP/IP) protocol suite. In 1983, the Department of Defense mandated that all DoD commands and agencies use these protocols for interconnecting computer systems over long haul facilities, and the Defense Data Network (DDN) was developed to provide wide-area service. At the same time, the internet protocol suite was integrated into the University of California Berkeley Software Distribution (BSD) of the UNIX operating system. Since BSD UNIX is the basis for many workstation and mini-computer operating systems, the internet protocol suite was rapidly propagated throughout the educational, scientific, and engineering community.

Today, the Internet protocol suite is the most widely used set of protocols for interconnecting heterogeneous systems. Hundreds of vendors offer products based on these protocols, including all of the major computer system manufacturers. Also, as rapidly as new transmission media and network technologies emerge, Internet protocol implementations are developed to support them. Virtually every form of local and wide-area transmission is supported, ranging from simple serial links to local area networks based on Ethernet, token-ring, and token-bus designs, to national and international packet-switched networks using terrestrial and satellite links. Even nodes operating over amateur radio links can be connected to the Internet.

Corporate Networking Technology

Computer networking technology has spread beyond governmental and educational institutions into the commercial business sector. As corporations shift the emphasis from centralized control to decentralized management of information resources, computer networking has become a critical component of a business' operation. With the availability of inexpensive workstations and personal computers, applications are becoming widely distributed and computing resources are becoming specialized and diverse. Coupled with the need to integrate all departments into the corporate network and to automate the business process, heterogeneous networking and interoperability have never been so important.

While proprietary network protocols can provide sophisticated and comprehensive services to a homogeneous community of systems, they are inappropriate for large networks of multi-vendor systems due to their lack of universal implementation. Therefore, the trend is away from proprietary networks and towards open networks that can interconnect a variety of computer systems. Use of open networks eliminates the reliance on a single vendor for services and equipment and makes it possible to rapidly adapt to changing technology.

OSI Reference Model

To properly understand the role of the API and the services it provides, a brief overview of the OSI Reference Model may be helpful. The Reference Model is defined by ISO International Standard 7498 (1984). Addendum 1 (1987) expands the definition to include connectionless-mode transmission.

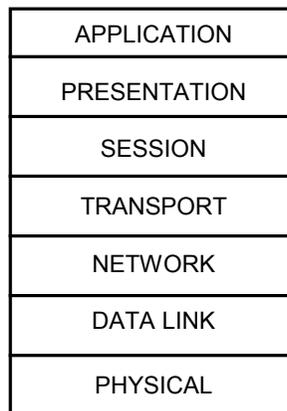
Open Systems Interconnection

Proprietary networks such as SNA tend to be closed, even though they may have been conceived as open. Each manufacturer develops its own set of conventions for interconnecting its own equipment and refers to these conventions as its *network architecture*. Recognizing the need to interconnect systems from many different manufacturers, ISO created a subcommittee with the objective of developing international standards required for Open Systems Interconnection (OSI). Their initial effort resulted in the development of a standard model of architecture that would constitute the framework for the development of standard protocols. This architectural model is known as the Basic Reference Model of Open Systems Interconnection.

The Seven Layer Model

The OSI Reference Model generally is depicted as a vertical stack representing the seven layers of the model. The following diagram shows this model:

The OSI Reference Model



The entire stack represents the OSI environment on the local system, and each layer represents a subsystem within the local system that implements the functionality of the given layer. The application layer is usually on the top, and the layer that represents the physical media is on the bottom. Systems that conform to the Reference Model are called open systems. Open systems are internally independent but provide common external services via standardized protocols publicly defined for each layer.

Layer Interaction

An entity operating within a layer (or subsystem) communicates with a peer entity operating within the same layer on another system, and using services provided by the lower layer, they cooperate to provide enhanced services to entities operating in the layer above. The basic idea of layering is that each layer adds value to services provided by the set of lower layers in such a way that the highest layer is offered services to run distributed applications. Layering thus divides the total communication process into smaller processes.

Another basic principle of layering is to ensure independence of each layer by defining services provided by the layer to the next higher layer, independent of how these services are performed. This lets changes be made in the way a layer or set of layers operates, provided they still offer the same service to the next higher layer.

Except for the highest layer, which operates for its own purpose, entities distributed among the interconnected open systems work collectively to provide specific services to entities in the layer above. Cooperation between entities within one layer is governed by protocols that precisely define how the entities work together using the services of the lower layer.

This list briefly describes the function of each layer and the basic services it provides to the layer above:

Application Layer (Layer Seven)

Provides a means for the application programs to access the OSI environment, and provides the distributed information service appropriate to the application.

The following are the basic application services:

- Identify and authenticate communication partners
- Determine adequacy of resources and quality of service
- Establish agreement on error recovery procedures, control of data integrity, and privacy mechanisms
- Select dialogue discipline and identify constraints on data syntax
- Provide synchronization and data transfer

Presentation Layer (Layer Six)

Provides the representation of information that application entities either communicate or refer to in their communication. This layer is concerned only with the representation of data (that is, syntax) and not its semantics.

Provides session services plus the following facilities:

- Negotiation and recognition of syntax
- Transformation of syntax, including data transformation, formatting and compression

Session Layer
(Layer Five)

Provides the means necessary for cooperating presentation entities to organize and synchronize their dialog and to manage their data exchange.

The basic session services are:

- Establish, synchronize, and release session connections
- Exchange normal and expedited data
- Manage dialogue interaction and report exceptions

Transport Layer
(Layer Four)

Provides transparent transfer of data between session entities and relieves them from any concern with the detailed way in which reliable and cost effective transfer of data is achieved.

The basic transport services are:

- Establish, maintain, and release transport connections (connection-mode)
- Transfer normal or expedited data (connection-mode)
- Map a request for transmission of a Transport Service Data Unit (TSDU) onto a request for the connectionless-mode network service (connectionless-mode)

Network Layer
(Layer Three)

Provides the means to establish, maintain, and terminate network connections between open systems, and provides transport entities independence from routing and relay considerations.

The basic network services are:

- Establish, maintain, and terminate network connections (connection-mode only)
- Provide sequencing, flow control, and expedited data transfer (connection-mode only)
- Select routes and quality of service
- Segment or block data and transfer network service data units
- Provide error detection, recovery, and notification

Data Link Layer
(Layer Two)

Provides functional and procedural means to establish, maintain, and release data link connections, or transfer data link service data units between network entities.

The basic data link services are:

- Establish, maintain, and release data link connection (connection-mode only)
- Provide sequence and flow control (connection-mode only)
- Delimit and transfer service data units over data link

Physical Layer
(Layer One)

- Provide error detection, recovery, and notification

Provides the mechanical, electrical, functional, and procedural means to activate, maintain, and deactivate physical connections for bit transmission between data link entities. This is the lowest layer, it provides the following basic services:

- Provides physical connection between data link entities
- Transfers physical service data units
- Provides fault detection and notification

International standards either have been or are being defined at each layer of the OSI Reference Model.

At least two standards are specified for each layer:

- A standard that defines the services provided by the layer.
- A standard that defines the protocol by which these services are provided.

A service interface standard at any layer frees users of the service from the details of how that layer's protocol is implemented, or even which protocol is used to provide the service, if more than one is defined. ISO International Standard 8072 defines the service interface for the transport layer.

The transport layer is important because it is the lowest layer in the OSI Reference Model that provides the basic service of reliable, end-to-end data transfer needed by application programs and higher-layer protocols. In doing so, this layer hides the topology and characteristics of the underlying network from its users. More importantly, the transport layer defines a set of services common to layers of many contemporary protocol suites, including Internet, ISO, XNS, and SNA. The protocol suite of primary interest to the API (Internet) is summarized in the following subsection.

Internet Protocol Suite

The maturity of the Internet protocol suite favors it as the dominant set of protocols for interconnecting multi-vendor systems. The protocol suite has been standardized by the U.S. Department of Defense, and certification procedures have been established to test protocol conformance. This suite of protocols is often referred to as the TCP/IP family of protocols, derived from the names of the transport and network layer protocols.

Major Application Layer Protocols

The Internet suite defines these major application layer protocols:

- TELNET virtual terminal service supports remote login from ASCII terminals. TELNET options are defined that permit login from full-screen, 3270-type terminals.
- File Transfer Protocol (FTP) supports the transfer of arbitrarily large files or programs between dissimilar file systems. File maintenance utilities such as renaming, deleting, and listing of file names are also supported.
- Simple Mail Transfer Protocol (SMTP) supports the exchange of electronic mail. SMTP can also be used to transfer files, but its use is generally limited to moderate size files containing ASCII text. Some systems may let SMTP be used to submit batch jobs and retrieve batch output.

Note: Presentation and session layer services are not defined as separate Internet protocols, and the corresponding functions are embedded within the application layer protocols.

Major Transport Layer Protocols

Internet defines these major transport layer protocols:

- **Transmission Control Protocol (TCP)** provides ordered, reliable transfer of a stream of data bytes with no explicit message boundaries. TCP is a connection-mode transport service that establishes virtual connections to associate two communicating processes.
- **User Datagram Protocol (UDP)** provides unreliable transfer of independent units of data called datagrams. UDP is a connectionless-mode transport service that maintains no permanent association between network processes. Since UDP maps directly into the underlying connectionless network layer, data transfer is very efficient.

Internet Protocol (IP)

(IP) is a sublayer protocol of the layer common to all connected systems. IP provides a connectionless datagram delivery service to any IP node on the internet.

If the destination node is not on the local network, IP routes the datagram to a gateway that forwards the datagram to the next network in the sequence of connected networks. Fragmentation and reassembly of datagrams is also provided by IP to support datagrams larger than the underlying layers can accommodate on the physical media.

Generally, IP is implemented over a Connectionless Network Service (CLNP), but connection-mode services such as X.25 are also used. In addition to wide-area packet-switched networks, local area networks based on Ethernet, token-bus, and token-ring technologies are generally used. These local area networks correspond to the IEEE 802.X series of protocols, usually without 802.2 (LLC) encapsulation.

OSI Terminology

The Unicenter TCPAccess API is oriented towards the Internet protocol, and can be described in terms of OSI-based, ISO protocols. Since the OSI is a general model, it is appropriate that OSI terminology be used to describe the operation and use of the API. This terminology applies to the Internet protocol suite as well. Sometimes the names are different, such as TSAP versus port for identifying upper-level entities, but, generally, the concepts are similar. This chapter describes concepts in both TCP and OSI terminology. After the initial reference to both terms, the book refers to concepts in TCP terminology.

The OSI architecture defines a set of rigorous conventions and terminology for specifying OSI-compliant protocols and service definitions. Some of this terminology has already been used in this document.

You have already been introduced to the fact that open systems are comprised of distinct, functional layers. Within each layer, entities communicate with entities above and below across an interface. For each layer of an open system, two standards must be defined: one describing the services provided by the layer and one describing the protocols used to provide the services.

Transport Services and Protocols

Since this document is concerned with the transport layer, it discusses transport services and transport protocols.

Transport Service Provider (TP)

The hardware and software that implement the protocols and services is called the *provider*. Thus, the provider of transport services is called the transport service provider.

Transport Service User (TU)

The hardware and software that uses the services of an adjacent layer is a user of those services and in this document is referred to as a transport service user.

Note: For brevity, sometimes refers to these terms as the transport provider and transport user, and simply abbreviates them as TP and TU.

In the most basic sense, the API implements the interface between a TU and a TP.

Service Access Points (SAP)

Each user of a given layer's services accesses the provider via a unique Service Access Point, generally abbreviated as SAP. A Transport Service Access Point is referred to as a TSAP. An identifier that specifies the location of an SAP is a Service Access Point Address, or simply *address* and if the identifier is that of an SAP in the transport layer, it is a *transport address*.

Connections

An association established by a layer between two peer users for the transfer of data is called a connection, and each terminal end of a connection within a service access point is called a connection endpoint. Thus, a connection between two transport users is called a transport connection, and its terminal end within a TSAP is called a Transport Connection Endpoint (TCEP), or transport endpoint for short. This document uses the term endpoint to mean a transport endpoint.

Primitives

The provider supplies services to the user via the invocation of primitives. A primitive specifies the function to be performed and is used to parameters consisting of data and control information. The primitives and their parameters indicate what information flows between the user and the provider of the service. The interface determines the method by which this information is conveyed.

These types of primitives are used to define the interaction between adjacent layers:

- Request** A primitive initiated by a user to request a service from the provider.
- Indication** An indication initiated by the provider to inform the user of some condition, such as the presence of incoming data.
- Response** Some indications require an action on the part of the user; this is the response primitive.
- Confirm** A primitive initiated by the provider to convey to the user that the information was given by the peer user to the peer provider in a response primitive. The confirm primitive implicitly means that the original request has completed.

Service Data Units

Finally, the unit of data transferred across the interface between a user and provider, and subsequently between two peer users, is called a Service Data Unit (SDU). The provider is responsible for mapping service data units onto Protocol Data Units (PDUs) by outgoing SDUs into PDUs and reassembling incoming PDUs into SDUs. A small service data unit whose transfer is expedited is called an expedited service data unit. Within the transport layer, these data units are referred to as TSDUs, TPDU, and expedited TSDUs, respectively.

Transport Layer Services

The transport layer is responsible for the delivery of data between transport users across a communications facility. In providing this service, the transport layer may need to deal with a variety of networks with differing characteristics and capabilities. Depending on the sophistication of the lower layers, the transport layer may require complex protocols to carry out its designated functions. By defining a set of common services and mapping these onto the capabilities and services of the network layer, the transport layer can shield upper layers from many of the details of data transfer. As such, the upper layers are able to focus on user requirements and applications.

While the services of any given layer are standardized, the method of acquiring them is not. This is because the manner in which information is conveyed between layers depends heavily on the environment in which they are implemented. Therefore, ISO 8072 defines the services of the transport layer and the API determines how these services are acquired.

The mechanisms used by the API to initiate, complete, and synchronize requests for service should be familiar to you as an MVS application programmer.

Note: The semantics of those requests, derived from the underlying transport services, may be foreign to you if you are unfamiliar with OSI-based networking. Therefore, the following section provides a brief overview of transport services and serves as an introduction to the concepts and facilities of the API. As before, this discussion is oriented toward OSI-compliant systems. However, any major differences between ISO and internet transport services are noted.

Modes of Service

The following services are supported by internet and ISO protocols:

Connection mode: A transport connection is established between two peer transport users prior to the exchange of data.

Connectionless-mode: No transport connection is established and each data unit transferred is independent of previous or subsequent data units.

Connection-Oriented Transport Service (COTS)

Connection-mode service within the transport layer is referred to in this document as Connection-Oriented Transport Service (COTS).

COTS provides the means to establish, maintain, and release transport connections providing duplex transmission between two transport users. It guarantees that all data units transferred arrive at their proper destination intact, uncorrupted, in the order in which they were sent. If this cannot be achieved, the user of the transport service is notified. COTS is the most widely used of the two modes of service.

COTS provides these basic services:

- Transport connection establishment
- Data transfer
- Transport connection release

These services describe the three phases of operation within the transport layer:

- The initial phase is the connection establishment phase, during which the transport connection is established
- The middle phase is the data transfer phase, during which all data is transferred
- The final phase is connection release phase, during which the connection is released and the association between transport users is terminated

COTS service primitives are many and varied and are discussed in terms of the basic services to which they apply. This table lists the COTS primitives associated with each basic service along with the parameters provided with each primitive.

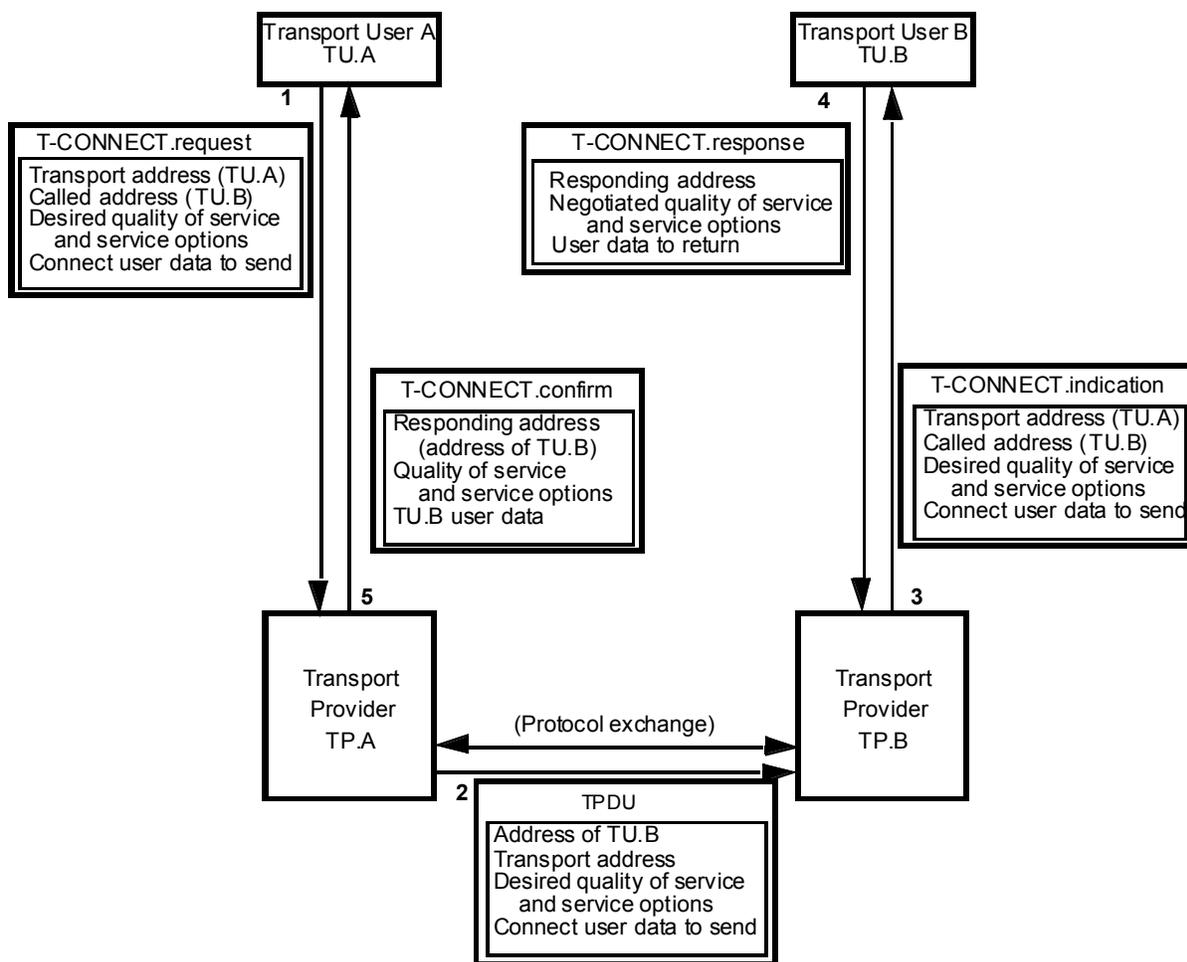
Service	Primitive	Parameters
Connection Establishment	T-CONNECT.request	Called Address Calling Address Service Options Quality of Service User Data
	T-CONNECT.indication	Called Address Calling Address Service Options Quality of Service User Data
	T-CONNECT.response	Responding Address Service Options Quality of Service User Data
	T-CONNECT.confirm	Responding Address Service Options Quality of Service User Data
Data Transfer	T-DATA.request	User Data
	T-DATA.indication	User Data
	T-EXPEDITED-DATA.request	User Data
	T-EXPEDITED-DATA.indication	User Data
Connection Release	T-DISCONNECT.request	User Data
	T-DISCONNECT.indication	Disconnect Reason User Data

Transport Connection Establishment

Transport providers establish connections to transport users, which the provider identifies by transport address pairs. The transport service and the users negotiate the quality of service of the transport connection. When connections are established, the class of transport service to be provided can be selected from a defined set of available classes of service. These service classes are characterized by combinations of selected values of parameters such as throughput, transit delay, and connection set-up delay and by guaranteed values of parameters such as residual error rate and service availability.

Connection Establishment Example

The connection establishment phase is best described by example. This diagram illustrates this process between two transport users, where Transport User A (TU.A) wants to communicate with Transport User B (TU.B):



The steps in this process are:

1. TU.A issues a T-CONNECT.request to its transport provider, TP.A, that includes this information:
 - Its own transport address (calling address)
 - The protocol address of TU.B (called address)
 - The desired quality of service and service options
 - Any connect user data to send to TU.B (connect user data is limited to 32 bytes)

At this time, TU.A may indicate whether the requested values for the quality of service and service option parameters are absolute or whether degraded values are acceptable.

2. The provider sends a transport connection request (TPDU) containing these items to its peer, TP.B.
3. TP.B issues a T-CONNECT.indication to TU.B.

If the T-CONNECT.request from TU.A indicated that degraded values for quality of service and service options was acceptable, either transport provider (TP.A or TP.B) or the peer transport user (TU.B) can negotiate particular parameters to a lesser value.

4. If TU.B wants to communicate with TU.A, it accepts the connection request by issuing a T-CONNECT.response giving this information:
 - The responding protocol address
 - Negotiated quality service and service options
 - Any user data to return to TU.A

Currently, the responding address given in the T-CONNECT.response primitive must be identical to the called address provided in the T-CONNECT.indication primitive. Future definitions of the transport service may let the responding address be different (e.g., the responding address may be the specific address resulting from calling a generic address).

5. After a suitable protocol exchange between TP.B and TP.A, TP.A issues a T-CONNECT.confirm primitive informing TU.A of the successful connection establishment. The confirmation message includes this information:
 - The responding address
 - Quality of service and service options
 - TU.B user data

The parameter values returned with the T-CONNECT.confirm primitive are the final, negotiated values that must be equal to or inferior to the requested values.

The TCP Connection Model

The OSI connection model requires that the transport provider obtain the explicit permission of the transport user to establish the connection. The TCP connection model differs from this in a subtle but significant way.

In the TCP model, the responding transport user is passive and does not intervene explicitly in the connection process. Rather, the transport user receiving connection requests advises its transport provider in advance which transport users it is willing to connect to. This is done by providing a partially or fully qualified protocol address of the calling transport user. Partially qualified protocol addresses are treated as wild-card specifications, and the null address indicates that a connection from any transport user is acceptable.

TCP also permits the establishment of a transport connection between two transport users that issue simultaneous connection requests. ISO does not. Simultaneous symmetric connection requests in the OSI model results in two transport connections. Generally, this is of academic importance, since the active and passive nature of clients and servers is such that simultaneous symmetric connection requests are avoided.

TCP connection service supports a type-of-service parameter that is defined at the IP layer and is used for the life of the connection. However, the definition of this parameter is not as rich and extensive as the ISO counterpart. Also, the results of negotiation are the opposite of ISO. If the two TCP transport users disagree on the choice of type-of-service, the superior value is used.

Data Transfer

The OSI transport layer offers these services:

- The transfer of normal data
- The transfer of expedited data

The expedited data transfer service must be requested during connection establishment and is negotiated as a service option.

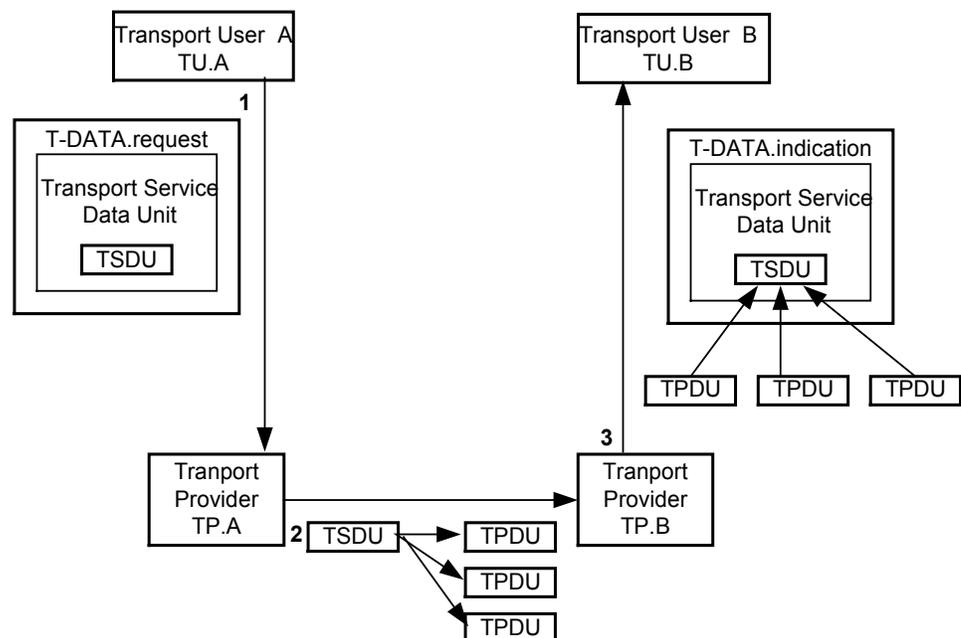
Two primitives are defined for normal data transfer.

The following diagram shows the data transfer process, which are:

1. TU.A issues a T-DATA.request to its transport provider (TP.A), providing the user data as a parameter. This user data parameter is the Transport Service Data Unit (TSDU) and may be of any arbitrary size.
2. The transport provider TP.A takes one of these actions and transfers the data to the remote transport provider (TP.B)
 - If the TSDU is longer than one TPDU, the transport provider segments the TSDU into multiple TPDU.
 - If the TSDU is shorter than one TPDU, the transport provider blocks it with previous TSDUs.

Note: In this example, transport provider TP.A segments the TSDU into multiple TPDU.

3. The transport provider TP.B must reassemble or unblock the TSDU and deliver it to the TU.B transport user using a T-DATA.indication primitive.



The TCP Data Transfer Model

TP.A and TP.B provide for the transfer of a stream of TSDUs in both directions simultaneously while preserving the integrity, sequence, and boundary of each TSDU.

TCP provides a similar service except that there is no notion of a TSDU. That is, TCP transfers a continuous stream of bytes with no record boundaries. If record boundaries are required by the transport user, they must be embedded within the data stream itself.

The ISO transport user has no direct control over when the transport provider forwards the data. However, the TCP transport provider can be forced into forwarding any buffered data by using a push mechanism. When invoked by the sending transport user, any data buffered by the local transport provider must be transmitted to the destination provider and delivered to the transport user. However, since there is no method for the destination transport user to determine where within the data stream a push occurred, this mechanism cannot be used to mark data boundaries. It serves only to force the transport provider to forward locally buffered data, using partially filled packets, if necessary. Pushed data is subject to the normal flow control restrictions of the protocol.

OSI Expedited Data Transfer Service

OSI also offers a data transfer service for small, expedited TSDUs containing up to 16 bytes of user data.

This service is optional in the sense that it must be:

- Requested by the calling transport user
- Agreed to by the called transport user
- Supported by the selected class of transport protocol

When this service is enabled, the sending transport user issues a T-EXPEDITED-DATA.request providing the data as a parameter, and the destination provider delivers the expedited TSDU as a parameter of the T-EXPEDITED-DATA.indication primitive.

The transfer of expedited TSDUs is subject to different quality of service and separate flow control from that applying to normal data transfer.

These are some of the key points:

- The transport provider guarantees that an expedited TSDU is not delivered after any subsequent normal or expedited TSDUs transmitted on the same connection
- Expedited TSDUs may bypass normal TSDUs but can be delivered to the destination transport user only when normal data is not being accepted

- Since expedited TSDUs are small, and since only one may be in transit at a time, expedited data transfer is best used for exceptional conditions and is inappropriate for bulk data transfer

The analogue in TCP transport service is called urgent data. It differs from ISO-expedited data transfer in that urgent data is transmitted in-band with normal data, and a pointer to where the urgent data ends may precede the data. The destination transport user is advised of the urgent condition and should dispose of unprocessed data quickly until the marked location in the input stream is reached. There is no length associated with TCP urgent data per se; the urgent pointer merely marks where the urgent data ends. Some providers attach special significance to the first byte of data following the urgent pointer and provide an out-of-band mechanism to read it. However, this is a nonstandard service.

Transport Connection Release

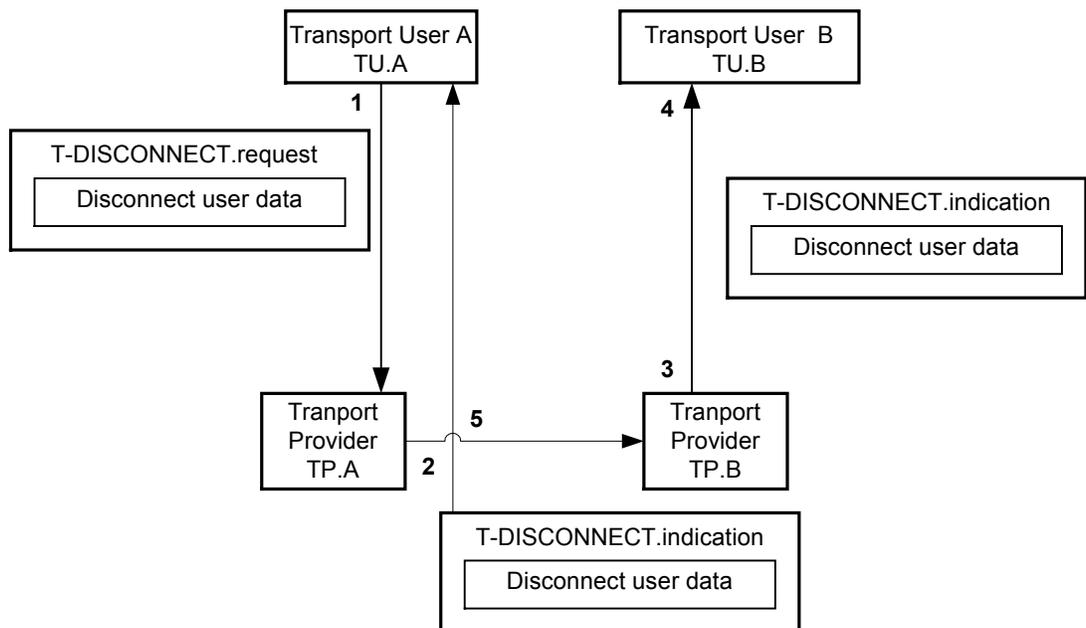
The connection release service is used to abandon connection establishment, release an established connection, or indicate a failure to establish or maintain a connection. Connection release is permitted at any time regardless of the current connection phase, and a request for release cannot be rejected. The transport provider does not guarantee delivery of any user data once the release phase is entered. Two primitives apply to connection release.

The following diagram shows the process the steps are required to complete as outlined below.

1. A T-DISCONNECT.request is issued to the applicable transport provider (TP.A) by either transport user (TU.A) to abandon connection establishment or to release an established connection.
2. The transport provider delivers a T-DISCONNECT.indication to the connected transport provider (TP.B).
3. The transport provider delivers the message to its transport user (TU.B). Up to 64 bytes of disconnect user data can be provided as a request parameter and is delivered to the destination transport user as long as the indication was the result of a user-initiated disconnect. A reason parameter is always provided with the indication.
4. The T-DISCONNECT.indication primitive also can be issued by a transport provider to indicate its inability to establish a connection or a failure to maintain an existing connection. In this case, no user disconnect data is provided with the indication, but a reason parameter indicates the cause of the indication (if known).
5. The ISO connection release service is similar to the TCP connection abort service (connection reset).

Note: Unlike TCP, ISO does not support an orderly connection release. This form of release guarantees that any buffered data is delivered to the transport user before the connection is released. ISO merely guarantees that buffered data is delivered to the destination transport provider.

The TCP orderly release service is sometimes referred to as a *graceful close*. Each transport user must agree to release the connection before the connection is dissolved, and until then, the TCP transport provider maintains the connection. Once a connection has been released by a transport user, no more data can be sent, but the user can continue to receive data. Any data previously sent is delivered to the destination transport user.



Endpoint States and Service Sequence

Four states are defined for the connection endpoint when using Connection-Oriented Transport Service (COTS). This table lists these states:

Note: Refer to [Connection-Oriented Transport Service \(COTS\)](#) for a list of valid sequences of COTS service primitives at a given connection endpoint.

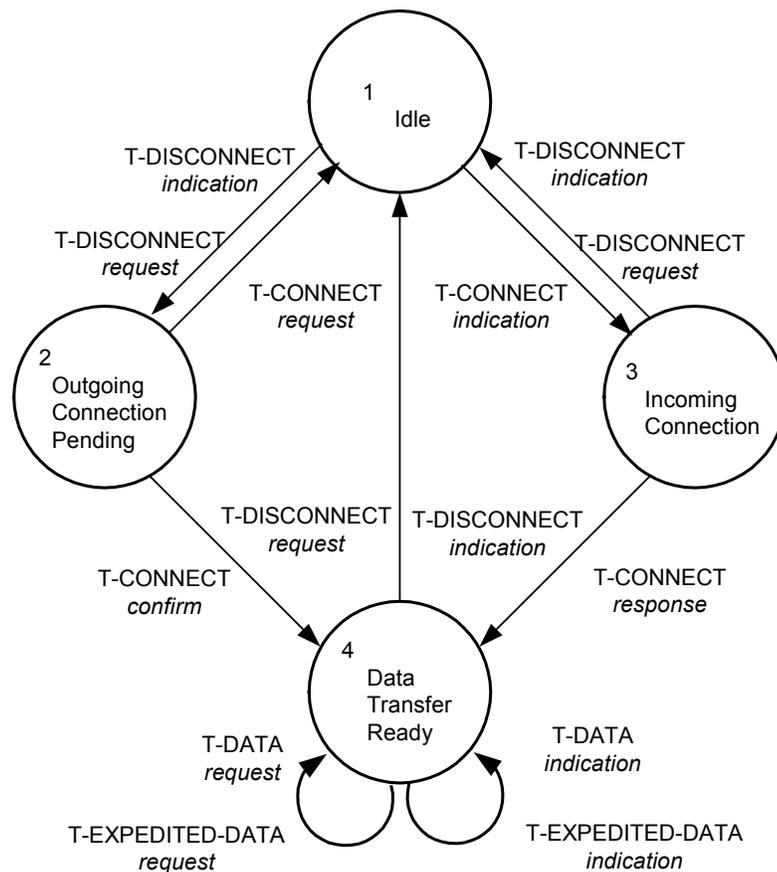
State 1 (Idle)	The idle state reflects the absence of a transport connection. It is the initial and final state of any sequence and once it has been re-entered, the connection is released.
State 2 (Outgoing-	The outgoing-connection-pending state indicates that connection establishment is in process and is entered when the local transport user issues a valid

- Connection-Pending) connection request. The transport user must wait for a connect confirmation or abandon the connection attempt with a disconnect request.

- State 3
(Incoming-Connection-Pending) The incoming-connection-pending state indicates that connection establishment is in process and is entered when the local transport user has received a connect indication from its transport provider. The transport user must either accept or reject the connection request.

- State 4
(Data-Transfer-Ready) The data-transfer-ready state reflects the presence of a fully established transport connection. End-to-end data transfer can only occur in this state.

This sequence diagram shows the COTS service primitives. The states are represented by circles and transitions between states are represented by arrows. The labels next to an arrow represent the events that can cause the state transition. The current state is at the base of the arrow and the new state is at the arrow point.



Connectionless Transport Service

The OSI Reference Model as originally published did not provide for connectionless-mode service. However, recognizing that this unnecessarily limited the power and scope of the reference model and excluded important classes of applications and network technology that are fundamentally connectionless in nature, an addendum was published that provided for connectionless-mode service within all layers of the model. Connectionless-mode service within the transport layer is referred to in this document as Connectionless Transport Service (CLTS). Within the Internet suite of protocols, connectionless-mode usually is referred to as a datagram service.

CLTS provides for the transfer of a single unit of data (TSDU) from a source TSAP to a destination TSAP without establishing a connection. The transport user can initiate such a transfer by the performance of a single service access. Since the transport provider is not required to relate this service access with any previous or subsequent access, all information required to deliver the TSDU must be provided with each access. This is in sharp contrast to COTS, where such information need be supplied only during establishment of the connection.

CLTS requires a prearranged association between peer transport users that determines the characteristics of the data to be transmitted, and no dynamic negotiation of parameters and options may be done. Since this prearrangement is unknown to the transport provider, such information must be provided with each unit of data to be transmitted.

From the perspective of the transport provider, data units are unrelated to their predecessors and successors, and therefore, ordered delivery without duplication or loss cannot be provided. No indication is returned to the sender when data units are delivered to the destination, and no flow control mechanism exists to prevent the source from sending data units faster than the destination can receive them. However, the transport provider does offer the optional guarantee that each data unit that arrives at the destination arrives intact—it is not truncated or corrupted. In the absence of congestion and network failures, CLTS can provide generally reliable service.

CLTS services are acquired via the invocation of two service primitives:

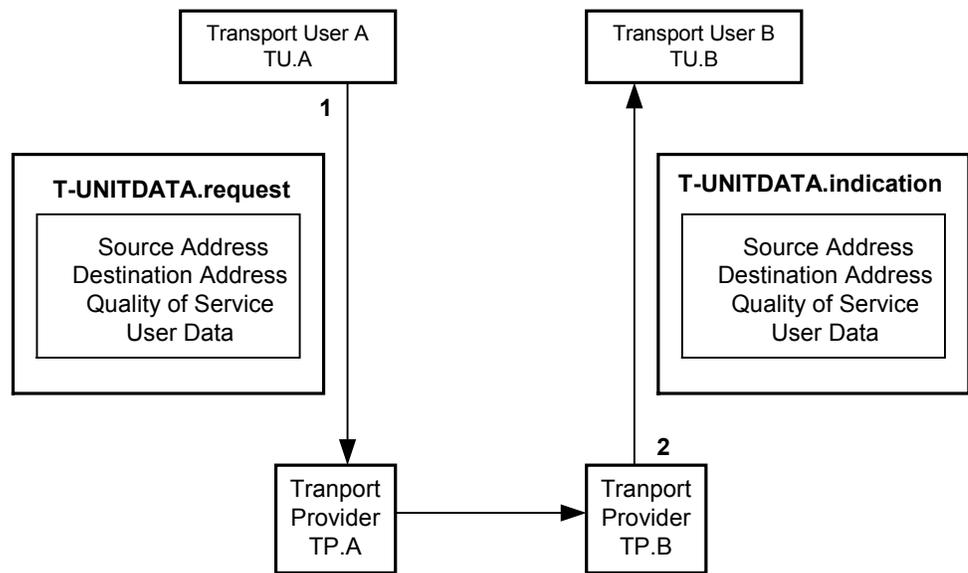
- T-UNITDATA.request
- T-UNITDATA.indication

Each primitive requires four parameters:

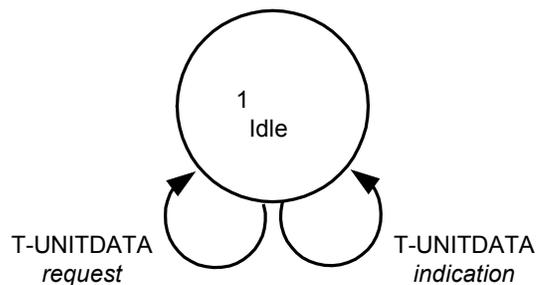
- The source transport address
- The destination transport address
- The quality of service
- The user data

The following diagram shows connectionless Transport service and demonstrates the following:

1. If a transport user (TU.A) wants to send a unit of data to some destination, it provides the data and all required parameters to the transport provider (TP.A) with a T-UNITDATA.request primitive.
2. When the transport provider (TP.B) receives a unit of data destined for the transport user (TU.B), it is presented to the user along with the accompanying parameters via a T-UNITDATA.indication primitive.



The following sequence diagram shows the simplicity of connectionless service for primitives issued at a CLTS endpoint:



Often the connectionless transport service is mapped directly onto a connectionless network service that has identical service primitives, resulting in a service that is efficient and easy to use.

UDP is the internet equivalent of ISO connectionless-mode transport service. UDP is similar in capability, except that UDP does not support the quality of service parameter.

Transport Layer Addressing

Many of the services described in this chapter require addresses to be supplied as parameters of the service primitive.

Example

COTS connection establishment primitives require the calling or called (responding) protocol address of the transport user, and the CLTS data transfer primitives require the source and destination protocol address of the TSDU. The addresses supplied with these primitives must identify the transport user, the transport provider through which the transport user is accessed, and the end-system (that is, host) in which the transport user and transport provider reside.

Internet Domain Addressing

Transport layer addresses for the internet domain contain a transport-layer and network-layer component. The transport-layer component is called a port number. At the network layer, the address is an internet address. Both the port number and internet address are fixed in length: four bytes for the internet address and two bytes for the port number.

An internet address is a 32-bit (four-octet) binary value. This value defines the overall address space that is a set of address numbers. The dot notation is a common way of representing an internet address so that users can read and write it easily. Each octet of the address is converted into a decimal number and the numbers are separated by dots.

The port number identifies services or clients using those services at a specified internet address. Server port numbers are generally used by multiple clients. Client port numbers are generally used by a single client and assigned dynamically by the transport provider.

Example

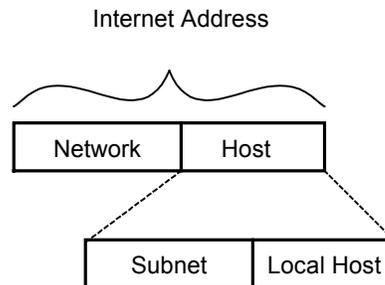
For MERMAID.HQ.MYCOMPANY.COM:

- The 32-bit binary address is 10001010 00101010 10000000 11010000
- The dot notation address is 138.42.128.208

An internet address is made up of two components:

- A network address
- A local (host) address

The following table describes the internet address components as illustrated in the following diagram.



Component	Part	Description
Internet Address	Network Address	Indicates the network to which the node is attached. Assigned by the Internet Network Information Center (NIC)
	Host Address	Identifies an individual node. Assigned by the authority that administers the particular network. The host number may be further subdivided into a subnet number and a local host address. This subdivision is administered by the network authority and is generally transparent to hosts outside of the network.
Host Address	Subnet	The network authority is responsible for assigning and administering subnet numbers. Unlike the network number, the subnet number does not contain any embedded information that would allow determination of its length. This information must be configured for each host connected to the subnet.
	Local Host	The network authority or subnet administrator is responsible for assigning local host numbers.

Network Classes The internet address is always four bytes in length, but the network and host numbers (addresses) vary in length according to the class of network.

Three network classes are currently in general use and are identified by the first two bits of the network number. The following table describes these network classes:

Note: The DECIMAL VALUE RANGE column indicates the decimal value the address starts with when the address is written in dot notation (for example, 138.42.128.208).

Class	Initial Bits	Description	Decimal Value Range
A	0xxx	Used for large networks. The NIC assigns a fixed value to the first byte of the address. The last three bytes are managed by the organization. The navy has a Class A address of 26, so all navy addresses are in this format: <i>26.nnn.nnn.nnn</i>	0 – 127
B	10xx	Used for medium sized networks The NIC assigns a fixed value to the first two bytes of the address. The last two bytes can be managed by the organization. Example: My Company has a Class B address of 138.42, so all internet addresses are in this format: <i>138.42.nnn.nnn</i>	128 – 191
C	110x	Used for small networks The NIC assigns one or more fixed values to use in the first three bytes of the organization's addresses. The organization has control of the last byte. Example: My Company has a Class C address of 198.137.88, so all internet addresses are in this format: <i>198.137.88.nn</i>	192 – 223

Connecting to an Internet Address

The transport user initiating a connection, or initiating a connectionless data transfer, must determine the address of the destination transport user in advance.

Generally, this information is acquired in one of these ways:

- The destination transport user is a standard service whose address is fixed and known throughout the network, such as a file server supporting a group of workstations on a local area network. Such addresses are generally referred to as well-known addresses.
- The destination transport user is not a well-known service, but its address is configured as part of the application program or is configured in a system database, and the application program can read it.
- A directory service or name server is used to map the name of a service or host into its address. The directory service can be located on the same host, in which case the transport user can access it directly, or it can be located on some remote host, in which case the transport user needs to know the address of the directory service.
- The user connects to a logger service whose address it knows and requests access to some particular service. The logger service spawns a process to service the user and returns its address so it can be connected to directly.

Sometimes a combination of these techniques is used.

Example

The well-known transport address may be known in advance and a name server is used to return the network address of a well-known service at some particular host.

It is generally not advisable to configure static information into an application program unless it is unlikely that the information will change. Therefore, the preferred approach is to use a dynamic service such as a network-wide directory service. In the internet environment, the Domain Name System (DNS) maps global names to addresses.

In either case, the user requests a local user agent to search the distributed directory for information associated with a globally-unique name. The local user agent then contacts a server agent and requests the desired information. If the information is available, it is returned to the requesting agent and forwarded to the user. Otherwise, the user agent might be redirected to another server agent, or the server agent might contact another agent on its behalf. This process continues until it is determined that the name is unknown, or a server agent is found that has the required information.

Concepts and Facilities

This chapter describes the basic functional concepts of the Unicenter TCPAccess Application Program Interface (API) and the facilities used to send requests to the API.

It discusses the following topics:

- [API Organization](#) – Describes the relationship between the API, the transport service provider, and the transport service user and the basic components and processing phases of the API
- [Concepts and Terminology](#) – Describes the fundamental concepts on which the API is based and includes definitions of applicable terminology
- [Establishing a Session with a Transport Provider](#) – Describes the processes involved in establishing a session between the API and a transport provider
- [Connection-Mode Service](#) – Describes the transport connection, endpoint management, and address management during a session between a transport user and its peer when the session occurs for an extended period
- [Connectionless-Mode Service](#) – Describes endpoint management and data transfer during short-term interaction between a transport user and its peer when the session occurs for an extended period
- [Connectionless Service with Associations](#) – Describes the interaction between two connectionless-mode transport users who are performing tasks other than simple request/response transactions
- [Local Endpoint Management](#) – Describes the API functions used to control processing at an endpoint
- [Declarative Macro Instructions](#) – Describes the macro instructions that are generally used to define data areas used by other macro instructions and which do not generate any executable code
- [Endpoint States and Function Sequences](#) – Describes the states that can occur at an endpoint and the functions that can be executed at an endpoint

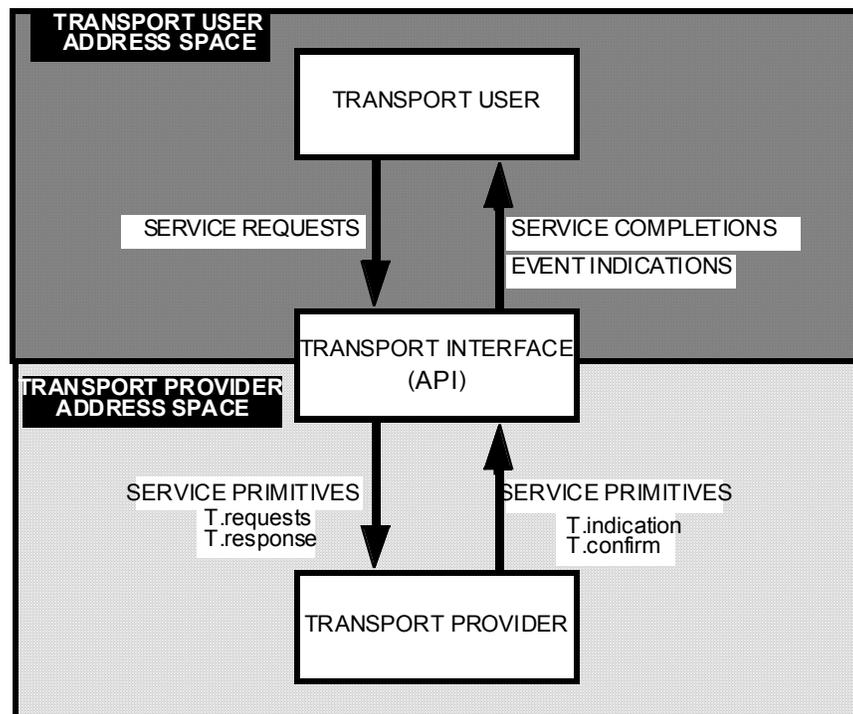
API Organization

The Unicenter TCPAccess API is an interface between a transport service provider (TP) and a transport service user (TU). The transport provider operates in its own address space and provides transport services to application programs or session layer entities using internet protocols. The transport user operates in its own address space and accesses those services using the transport interface (the API).

Relationship Independence

The transport user need only be concerned with the characteristics and facilities of the API and can operate without knowledge of the underlying protocols. Regardless of the transport protocol, the procedures used to establish a connection and transfer data are generally the same. Because the transport layer hides the details of lower layers, application programs using the API are independent of protocol as well as the physical medium over which communications occur.

The following diagram shows the relationship among the Transport User, Transport Provider, and API.



The transport user issues requests for service that are mapped by the interface into service primitives supported by the transport provider. For an OSI-compliant provider, a request or response primitive is the result. Similarly, service primitives issued by the transport provider and received by the API are mapped into service completions or special event indications. In this case, the OSI provider issued an indication or confirm primitive.

Service Request Processing

Service requests issued by the transport user involve these phases of processing:

- Phase 1 Initial processing performed during invocation of the request.
- Phase 2 Primary processing performed within the transport provider's address space.
- Phase 3 Final processing performed in the transport user's address space when the request has been completed.

The amount of time required to complete a request can range from immediate to indefinite, depending on the type of request and the current processing load on the system. Requests that require protocol exchanges between transport layer entities also can be delayed by network malfunctions or congestion in the network.

Service Request Modes

The following modes are provided for processing service requests:

- Synchronous** Blocks the transport user until the request is completed. In this case, the task issuing the request is suspended and is not redispached (except to process asynchronous interrupts) until posted during the final processing phase.
- Asynchronous** Does not block the transport user when a protocol event must occur in order to complete the request. Control is returned to the issuing task as soon as this is determined. When the necessary events occur, processing in the transport provider's address space is performed asynchronously, and the transport user is informed when the request completes. In the mean time, the transport user can perform other tasks, including issuing more requests to the transport provider.

Instead of anticipating an event by issuing a request and waiting for it to complete (that is, issuing a receive request to read some incoming data), the transport user might prefer to be signaled when certain events occur, and then respond by issuing the appropriate request. When properly implemented, this method also prevents the transport user from being suspended. In this case, special event indications are signaled to the transport user by scheduling exit routines that can preempt normal processing.

API Components

The API operates as an MVS subsystem and consists of several components. This list of components starts with the transport provider and works up toward the transport user.

Transport Provider
Interface Routines

Interface routines that run in the transport provider's address space under control of a space-switch PC routine or a transport provider dispatchable unit (for example, the transport provider's address space must be dispatched to run these routines).

These routines provide protocol request event handling on behalf of the transport user.

Infrastructure (IFS)

An infrastructure that provides the cross-memory environment required for communicating between address spaces.

This component consists of routines to initialize the subsystem and cross-memory environment, routines to manage resources used by the transport interface, and MVS subsystem exits to perform cleanup during task and address space termination.

Space-Switched
Program Call (PC)
Routines

Routines located in the transport provider's address space that are executed under control of the transport user's TCB (for example, the transport provider's address space is not dispatched to run these routines).

PC routines let code and data structures be located outside of the transport user's address space, thereby maximizing the amount of local storage available for use by both the application program and the transport provider.

Transport User
Interface Routines

Interface routines located in common storage that can be called directly by the transport user and that preprocess requests before forwarding them to the appropriate PC routine.

Placing these routines in common storage eliminates the need to link edit any API routines with the application program or to load any executable modules at runtime. Some of these routines also run asynchronously to process event indications issued by the transport provider.

Macro Instruction Library	A library containing macro instructions that generate parameter lists and user data structures required by API routines that generate the linkage to common storage interface routines to initiate service requests.
C Service Functions Library	<p>A library of functions that provide services similar to the assembler language interface for programs written in the C programming language.</p> <p>These library routines let you generate API service requests as standard C function calls. The data structures used are identical in format and content to those used by the assembler language interface.</p>
Socket Interface Functions Library	<p>A library of functions that implement a socket interface for application programs written in the C programming language.</p> <p>The API socket library enhances portability of networking applications developed to run on BSD UNIX systems or systems supporting the BSD socket interface.</p>

Concepts and Terminology

This section introduces the fundamental concepts on which the API is based. This section also defines important terms used throughout the remainder of this programmer's reference. This terminology is the real instantiation of the abstract Open Systems Interconnection (OSI) terminology introduced in the chapter "API Overview."

Modes of Service

The API supports connection-mode service (Connection-Oriented Transport Service (COTS)) and connectionless-mode service (Connectionless Transport Service (CLTS)).

Connection-Mode Service	A connection is established for the purpose of transferring data. All data is delivered intact, uncorrupted, in the same order as transmitted and without duplication.
-------------------------	--

Connectionless-Mode Service No connection is established and a unit of data can be transmitted to any destination, or be received from any source. Each data unit is independent of previous and subsequent data units, and no guarantee is made with regard to the reliable delivery of data. The API also supports a hybrid of these two service modes called *associations*. Associations are to CLTS as connections are to connection-oriented transport service (COTS): a long-term binding between two connectionless transport service (CLTS) users for transferring data. Associations let the transport user use a connectionless service in a connection-oriented fashion. This feature is transparent to the provider and implemented entirely within the transport interface.

Endpoints and Access Points

Services can be acquired only through transport user endpoints. Services use access points to address transport users.

Endpoints A COTS endpoint represents the TU end of a connection and is the source or destination of all data transferred via the connection.

A CLTS endpoint simply represents a transport user (TU) source and destination of connectionless data units. An endpoint can be thought of as a logical channel of communication between the transport user and transport provider. A transport user may use multiple endpoints, and service interactions with the provider are multiplexed and de-multiplexed based on the endpoint with which they are associated

A transport provider creates and dissolves endpoints at the request of the transport user. The process of creating an endpoint is called opening, and the process of dissolving an endpoint is called closing. When an endpoint is opened, it is given a unique identifier that must be provided with each subsequent service request associated with the endpoint. This identifier is called an endpoint ID. When opening the endpoint, the transport user specifies the communications domain within which the endpoint exists and the service mode desired. The API uses this information to determine the transport provider and protocol that services the endpoint.

Access Points The transport service access point through which the transport user is addressed must also be declared. This is done by binding the ISO TSAP address (transport address) or TCP port number to the endpoint. The transport address can be specified by the user or assigned by the provider. The service access point can be changed later without closing the endpoint by runbinding the current transport address and then binding a new one.

Each endpoint can be bound to a unique transport address, or multiple endpoints can be bound to the same address. An endpoint cannot, however, be bound to more than one address. Whether the relationship between endpoints and access points is one-to-one or many-to-one is generally determined by the connection strategy employed by the transport user.

Connection Strategies

The binding service is also used to indicate how many connect indications can be pending to the transport user. If this value is greater than zero, the endpoint is said to be enabled. Otherwise, the endpoint is disabled. CLTS endpoints not engaged in an association are always disabled, and once the local transport address is bound, are ready for sending and receiving data. COTS endpoints cannot engage in data transfer until connected.

One of two connection strategies can be used by the transport user. The determination of which strategy to use is based on the role of the transport user and its relationship with other transport users with whom it connects. Typically, one user is the provider of some service, and the other is a consumer.

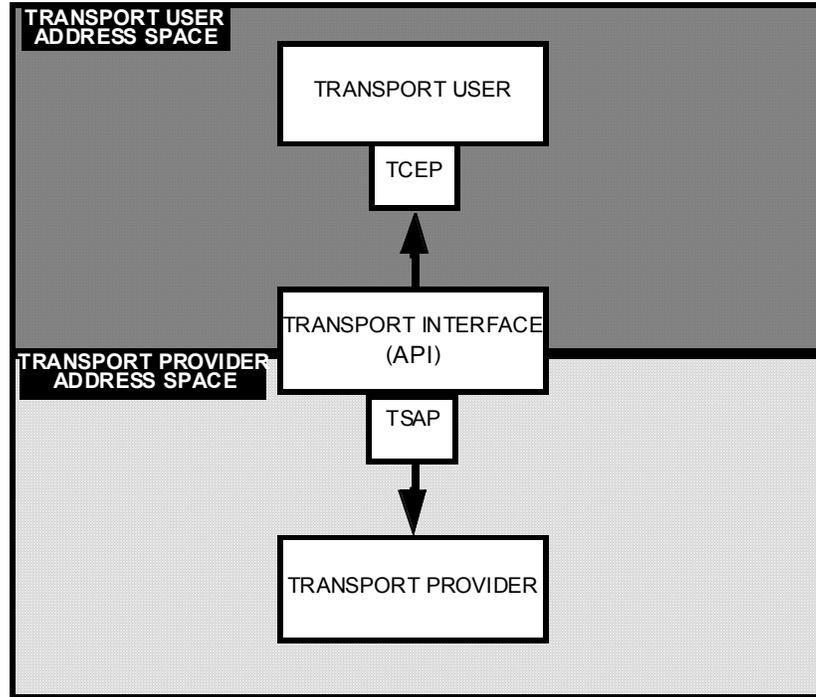
The transport user providing a service (the server) operates in client mode.

Client Mode

The client is the active participant in establishing a connection. The client TU initiates the connection establishment phase by issuing a request to connect to the server. The client TU then waits for confirmation that the connection has been established. Client-mode requires a disabled endpoint. When the connection is confirmed, the client TU can enter the data transfer phase.

Client-mode is generally characterized by a one-to-one relationship between endpoints and access points.

The following diagram shows client mode with one endpoint per TSAP.



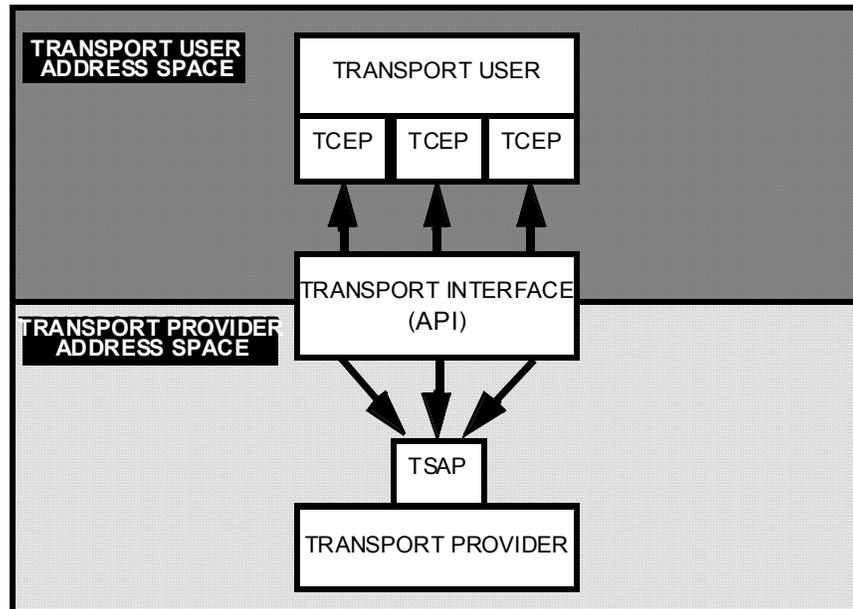
Server Mode

The server TU is the passive participant and listens for connect indications generated by the transport provider. This requires an enabled endpoint. When a connect indication is generated, the server must decide whether to accept, establish the connection or reject (abandon) the connection.

If the connection is accepted, the endpoint enters the data transfer phase and a confirmation is returned to the client.

Single-Threaded Servers are transport users operating in server mode that service their clients one at a time. When a single-threaded server is connected to a client, no other clients can be serviced. This mode is used when the service can be performed in a relatively short period of time or when the role of the two connected users is not clearly distinguished, and by prearrangement one agrees to act as the server. The latter situation is a consequence of the connection model requiring one party to initiate and the other party to respond. An analogy of this situation is a telephone system that requires each party to call the other at precisely the same time whenever one desires to talk to the other. Such a system is unworkable. Therefore, one initiates the call and the other answers. The previous diagram also shows the client/server relationship in a single-threaded server.

Multithreaded Servers can service many clients simultaneously and are typical of the traditional client/server model. Since many connections can be active at one time, multithreaded service is characterized by a many-to-one relationship between endpoints and access points.



Working in multithreaded environments adds complexity to the connection strategy. Because the endpoint on which a connect indication arrives must be available for receiving additional indications, the connection must be established to a new endpoint. Typically, a multithreaded server reserves one endpoint for receiving connect indications and establishing connections to new endpoints, all bound to the same access point.

Client/Server
Connection

Another characteristic of clients and servers is that clients must know the address of the server in advance. Therefore, the server access point is generally at a well-known address.

When the client TU initiates the connection request, the server TU is given the address of the caller. Since the server does not need this address in advance, the client can use an endpoint that is allocated dynamically. Often, the client lets the transport provider assign the address of its access point.

Once the connection is established and the data transfer phase is entered, the distinction between client and server is usually unimportant, relative to the transport provider; the client TU and server TU send and receive data in exactly the same manner.

Data Transfer Modes

When a connection is established, the API may transfer data in two modes, Transport Layer Interface (TLI) mode and Sockets mode. TLI and Sockets modes only affect data transfer; there are no connection issues associated with them. The transfer mode is specified in the `TOPEN` macro, and data transfer primarily affects the `TSEND` macro.

TLI Mode

The default data transfer mode—used in all previous releases of Unicenter TCPaccess. TLI mode is a programmable interface that lets applications be built independent of the networking protocols below them, and provides reliable network transmission.

A COTS data send request in TLI mode completes when all of the data is acknowledged by the remote transport provider. A CLTS data send request completes when the data is given to the network. The amount of data that can be sent is subject to limiting values defined by the installation or negotiated by the transport user. A TLI mode data send request is *all or nothing* with respect to the amount of data to be sent. If all data cannot be sent, none of it is sent and the request completes in error.

Socket Mode

Socket mode data transfer feature was introduced with Version 5.2 of Unicenter TCPaccess. When the API is using Socket mode, data transfer operates in a manner similar to BSD sockets.

Socket mode allows data send requests to specify larger amounts of data than can be accommodated by the currently available send buffer space. In TLI mode, if you exceed the transfer data window size, an error occurs. In Socket mode, data transfer continues as the send window reopens to continue. A `TSEND` or `TSENDTO` request in Socket mode completes when all of the data that will be sent is passed to the local transport provider (for example, TCP or UDP).

The amount of data that is actually sent depends on whether the request specifies blocking or non-blocking operation (specified by `OPTCD=BLOCK` or `OPTCD=NOBLOCK`). If blocking is in effect, then all of the data in the send request is sent. If non-blocking is specified, all, some, or none of the data may be sent. The amount of data that is actually sent depends on the space available in the current send buffer.

Disconnect and Orderly Release

When two connected transport users complete data transfer, the transport connection can be released in one of two ways:

- Disconnect Service
- Orderly Release Service

The method used depends on the characteristics of the transport user as well as the capabilities of the transport provider.

Disconnect Service

The simplest method commonly supported by all transport providers. When a connection is released in this fashion, the termination is abrupt, and any previously transmitted data not received by the connected transport user may be discarded.

If the transport user is a session layer entity, coordinated release and forwarding of user data is handled by the session layer protocol. Otherwise, the application program must implement its own procedures for coordinated release of the connection or accept the consequences of a possible loss of data.

Orderly Release Service

Alternate method implemented by the transport provider. This service provides for a graceful release of the connection that does not occur until both transport users agree. If each transport user receives all buffered data before agreeing to the release, no data is lost.

Service Requests and Parameters

The transport user issues service requests to the API by executing transport service functions. Each transport service function has a corresponding assembler language macro instruction that generates a parameter list and calls the appropriate API routine to execute the function. Application programs written in the C programming language use a runtime library of C functions that provide the necessary assembler language interface to the API routines

A function code passed to the API routine identifies the requested service, and a parameter list contains all other information needed to execute the function.

The parameter list can be generated in-line or out-of-line to support both reentrant and nonreentrant programming. This parameter list is formally known as a Transport Service Parameter List (TPL). It serves the same purpose as an RPL in VTAM.

Note: The TPL contains many parameters found in the RPL, such as option codes, ECB or exit routine addresses, return codes, and so forth.

The TPL is the primary structure for requesting API services and exchanging information. Refer to [The Transport Service Parameter List](#) for a detailed description of the TPL.

The TPL contains these types of information:

- Parameters and control information common to all functions and used primarily to coordinate execution with the transport user
- Function-specific parameters that are fixed in length and can be stored within the TPL
- Function-specific parameters that are variable in length and whose value is stored in a user-provided storage area

The TPL can be generated or manipulated in a variety of ways. The simplest method is to use the in-line form of macro instructions that build the TPL in line with assembler instructions to initiate the request. In-line macro instructions build a separate parameter list for each macro instruction. Since many service requests require similar information, it is more efficient to reuse the same TPL for other types of requests. A list form of each macro instruction is provided for this purpose.

The in-line and list forms have the characteristic of being nonreentrant. If the transport user is reentrant, the TPL must be built dynamically in local storage. Therefore, there are forms of each macro instruction that generate, modify, or execute TPLs in a storage area designated by the transport user. An assembler language dummy control section (DSECT) is also provided so the user can build and manipulate the TPL directly.

Using one of these methods, the transport user constructs a parameter list containing the necessary information and initiates its execution. The API interprets the information in the parameter list to determine the service (that is, function) requested, validate the parameters, and schedule the request for further processing. When processing is complete, the parameter list is updated and any information to be returned to the transport user is stored in the designated storage areas.

Before discussing the transport service functions supported by the API, it may be helpful to introduce some of the parameters that affect their processing. You will find a more thorough discussion of service parameters in the *Unicenter TCPaccess Communications Server Assembler API Macro Reference*, which describes the operands of each macro instruction in detail.

Common Parameters Are present in all requests for service. They represent the smallest subset of the TPL required to execute a transport service function. Generally, this information is used to coordinate processing with the transport user.

The following is a list of common parameters:

- A TPL identifier
- The function code
- A semaphore indicating whether the parameter list is in use
- Various flag bits affecting execution of the request
- The endpoint identifier
- The address of an ECB to post or an exit routine to schedule when the request is complete
- Option codes that modify execution of the request or indicate special conditions
- Return codes that indicate the success or failure of the request and determine error recovery actions

All requests are associated with some endpoint and require an endpoint identifier. The one exception is the open service, which requires an endpoint identifier of zero for opening a new endpoint.

Option codes are used to alter execution of a request by selecting optional facilities or indicating special conditions (for example, indicating the end of a TSDU or selecting a synchronous or asynchronous mode of execution). Option codes are interpreted only by the API and should not be confused with protocol options, which are processed by the transport provider. The latter are specified with a separate parameter as described in [Transport Protocol Options](#).

In the following sections, some of these option codes are referred to by name. For example, the option code that indicates the end of a TSDU is called EOM (End of Message). Writing OPTCD=EOM is the notation for signifying that the EOM option code was, or should be, indicated. Option codes usually come in pairs where one code indicates the opposite condition of its counterpart. In the example, NOTEOM is used to indicate the continuation of a TSDU.

The remaining common parameters are used primarily for synchronization and the handling of errors and exceptional conditions. Refer to the chapter “Program Synchronization and Control” for detailed information.

Fixed-Length Parameters

Fixed-length parameters are function-specific parameters whose value can always be stored in a 32-bit word. They are passed by value (that is, stored within the TPL itself), and generally specify information only of interest to the API or the transport user.

This is a list of some examples of fixed-length parameters:

- Sequence numbers identifying pending connect indications
- Endpoint identifiers for accepting connect indications to a new endpoint
- TCB and ASCB addresses used for passing ownership of endpoints
- The size of the connect indication queue
- Residual byte counts after send and receive requests
- Disconnect reason codes
- Datagram error code

Fixed-length parameters are used for returning information as well as supplying information. There is room in the TPL for three such parameters. The number of parameters used is specific to each function, as is the type of value stored in a given parameter location. Parameter locations unused for a given function are automatically cleared by the API.

Variable-Length Parameters

Variable-length parameters are passed by reference. In this case, the address and length of the parameter is stored in the TPL and the parameter itself is stored in the indicated storage. In almost all cases, these parameters contain information that is exchanged directly between the user and provider and indirectly between peer users. Often this information is not interpreted by the API and is merely transferred from one to the other.

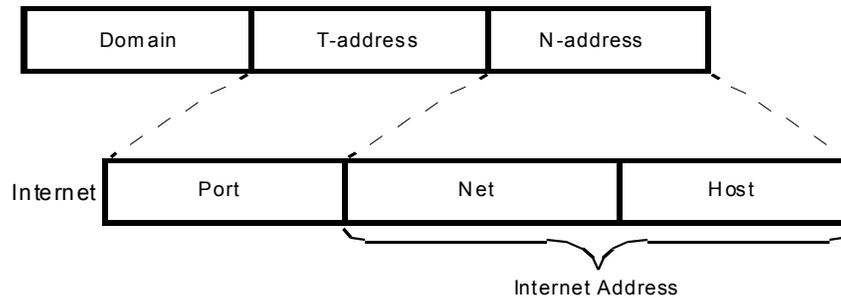
These three variable-length parameters are defined:

- Transport protocol address
- Transport user data
- Transport protocol options

Variable-length parameters are the primary arguments of the transport interface. While the value of these parameters may be provider-specific, their use and relationship to the transport interface is not.

Transport Protocol Addresses or protocol addresses for short, contain the addressing information necessary for establishing connections and identifying the source and destination of connectionless data units.

Protocol addresses consist of three components.



The structure of a protocol address is defined for program access and manipulation by assembler language DSECTs and C structure declarations available to the application program. The assembler language DSECTs can be found in the appendix, "Data Structures" of the *Unicenter TCPaccess Communications Server Assembler API Macro Reference*. The equivalent C structure declarations may be found in the chapter "C Language Structures" of the *Unicenter TCPaccess Communications Server C/Socket Programmer's Reference*.

The domain field of the protocol address identifies the communication domain to which the address belongs. This field must match the communication domain specified when the endpoint was opened. In particular, this field identifies the address as belonging to the Internet domain.

The two remaining fields contain a transport address and a network address, respectively. In the Internet domain, this is a port number and internet address. Addresses in the Internet domain are fixed in length: a two-byte domain identifier followed by a two-byte port number followed by a four-byte internet address, for eight bytes.

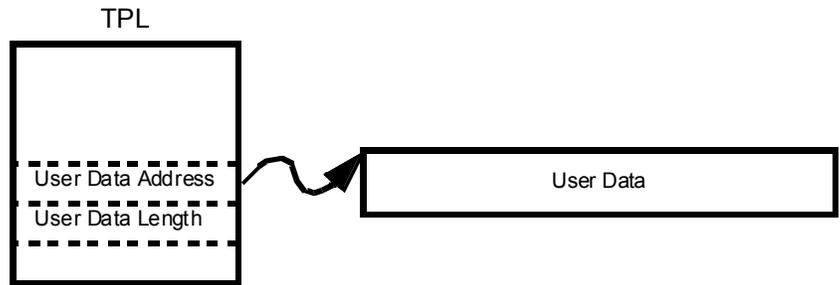
Sometimes it is permissible to supply a partial protocol address. For example, when binding the endpoint to a local access point, it is generally not advisable to provide the network address because the transport provider already knows its network address. In cases where the host is connected to more than one network, it may have multiple network addresses and the correct one cannot be determined until certain routing decisions are made based on the destination address. A partial protocol address is specified by indicating the absence of the network address, by either its length or its contents set to zero.

Transport User Data is referred to simply as *user data* throughout the remainder of this guide. User data is usually that data exchanged between transport users using send and receive requests.

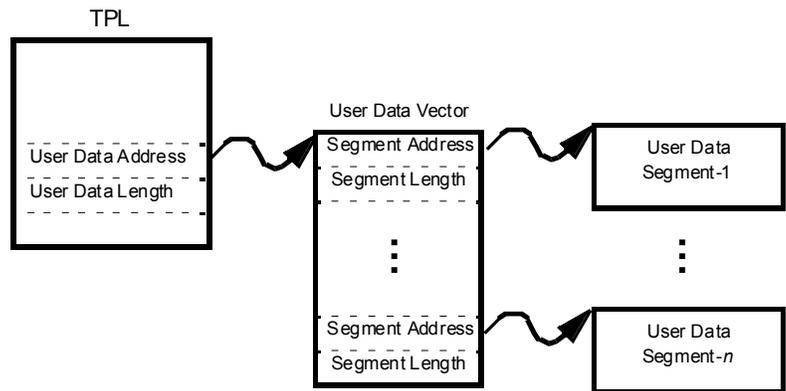
User data is an arbitrary string of data bytes, uninterpreted by the API or the transport provider. No particular character format is assumed (that is, the data may be ASCII, EBCDIC, or pure binary data). The only restriction is the amount that can be transferred with a single service request, and the data must consist of imposes syntax on user data transferred between transport entities. In the absence of an OSI protocol stack, the application program has sole discretion over the content of user data. User data generally is supplied or returned as a single, contiguous string of bytes. This is called direct mode, because the user data parameter directly identifies the data. Sometimes it might be more convenient to send or receive data as noncontiguous segments. This often is referred to as scatter-read or gather-write, or, more simply, *indirect mode*, because one level of indirection is required to locate the data. In this case, the user data parameter references a vector, each element of which defines a segment of contiguous data.

The following diagram shows direct user data parameters and indirect user data parameters.

Direct User Data Parameters



Indirect User Data Parameters



Transport Protocol Options

Transport protocol options are used to:

- Enable optional facilities
- Specify certain service parameters
- Override installation defaults for selected internal variables (for example, specifying quality of service parameters, enabling expedited data service, and modifying internal buffer parameters)

In almost all cases, protocol options apply only to the transport provider or transport user and are passed through the API without interpretation. The format is provider-specific. To assist the application program in manipulating these options, assembler language DSECTs and C language structure declarations are provided. Protocol options are provider or protocol specific.

In one case, the protocol options parameter is used to specify or change the API variables. These variables affect how many send or receive requests can be pending for an individual endpoint, and how much buffer is allocated. These are not exactly protocol options, even though the protocol option parameter is used to manipulate them. An option code is provided to indicate whether the protocol options parameter contains API options or transport provider options.

The Transport Service Parameter List

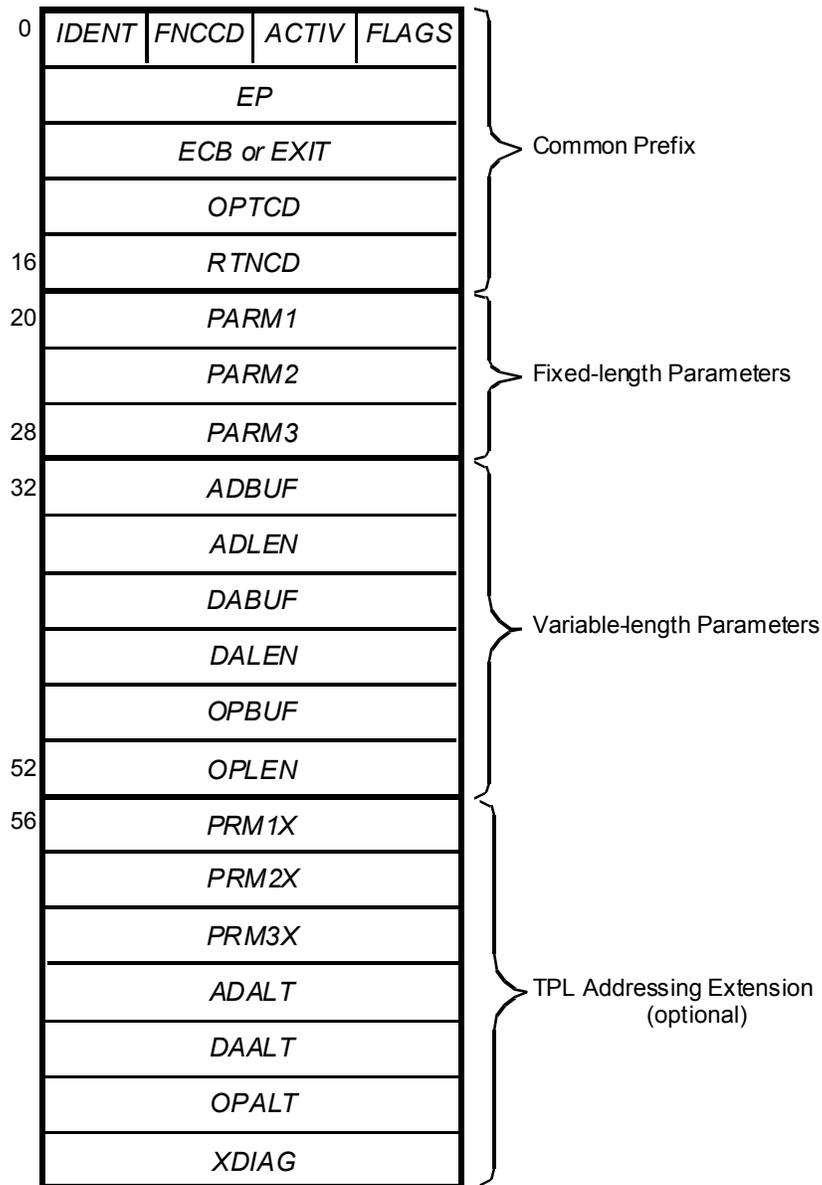
The parameter groups reflect the physical structure of the TPL. The standard TPL is 56 bytes in length and must be aligned to a fullword boundary. If `OPTCD=EXTEND`, the length is 84 bytes. A dummy control section is provided to map the fields of the TPL and can be found in the *TCPaccess Assembler API Macro Reference*.

The TPL is organized with the most frequently-used information at the start. Since many transport service functions only interpret a subset of the TPL, a shorter version can often be used. The in-line form of macro instructions generates a short TPL unless forced to do otherwise. All forms of API macro instructions support a short and long version of the TPL. Use caution, however, as this feature is function-specific. Refer to the *TCPaccess Assembler API Macro Reference* for detailed information. The name given to a field within the TPL corresponds to the macro instruction operand that references that field. An extended version of the TPL is available as well. Extended TPLs append additional fields to the long form TPL that may contain ALETs of ESA extended addresses.

Example Option codes indicated with the OPTCD operand are stored in the OPTCD field. The symbol defined by the TPL DSECT that corresponds to this field is constructed by prepending TPL to the name of the field. Therefore, the option codes are stored at the symbolic location TPLOPTCD.

Some of the examples used in the remainder of this chapter contain references to symbols defined by the TPL DSECT.

TPL Standard Format The following diagram shows the standard format of a TPL containing a common, fixed-length, and variable-length parameter section.



Common Prefix

The TPL starts with a common prefix present for all transport service functions. The parameter list prefix consists of control information, completion status, option codes, and the endpoint identifier. It also contains the minimum amount of information required to execute a transport service function.

This list describes the major fields within this prefix.

IDENT Identifies the data structure as a TPL. A version number can be encoded within the identifier to indicate which version level of the API generated the TPL in the event future versions change the use of certain fields.

Note: This field should not be modified or interpreted by the application program.

FNCCD Contains a function code that designates the transport service being requested. This code usually is set automatically by macro instructions corresponding to each transport service function.

ACTIV This byte contains the test-and-set semaphore used to indicate whether the TPL is active. This field is cleared by the TCHECK macro instruction when the requested function completes. The TPL must not be modified while it is active.

Note: The application should never set or clear this field directly.

FLAGS Contains flag bits set and cleared by the API and generally is of no interest to the application program.

EP Contains the endpoint identifier of the endpoint associated with the request. Except for TOPEN, this field must always contain a valid endpoint identifier of a currently opened endpoint. However, this field can be changed between function requests to reference different endpoints.

ECB or EXIT Used for synchronization and is shared by these different uses:

- An internal ECB
- The address of an external ECB posted when the function completes
- The address of an exit routine entered when the function completes
- The FLAGS field is used to indicate how this field currently is being used

OPTCD Contains option codes controlling how a function request is processed. This field is used primarily by the application program to provide information to the API. However, in a few cases, the API may set bits to indicate special conditions that have occurred. OPTCD field format is composed of four consecutive bytes.

OPCD1	OPCD2	OPCD3	OPCD4
-------	-------	-------	-------

- OPCD1 specifies options that apply to all transport service functions
- OPCD2, OPCD3, and OPCD4 specify options that apply to specific functions or groups of functions

Once specified, options remain in effect until explicitly changed. A zero value for any bit or subfield represents the default indication of the corresponding option code.

RTNCD The field in which the API returns completion status. RTNCD is a fullword consisting of these subfields:

ACTCD	ERRCD	DGNCD
-------	-------	-------

- ACTCD** A recovery action code used to determine the appropriate action on completion of a transport service function.
- ERRCD** A conditional completion code or specific error code, depending on whether the function completed conditionally or abnormally.
- DGNCD** Contains a module and instance code to identify the specific instance of the error.

Fixed-Length Parameters

Immediately following the parameter list prefix is a section consisting of three fullword, fixed-length parameters. These parameters are function-specific and are passed by value; that is, the parametric value itself is stored in the TPL.

This list describes the fixed-length parameter fields.

PARM1 Shared by several uses. Generally, it contains a parameter provided by the application program that is processed by the requested function. The API also can use this field to return a parameter to the application program. The alias names QLSTN, SEQNO, and TCB are used to reflect the function-specific use of this field.

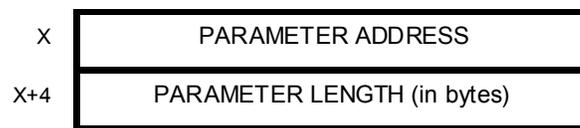
PARM2 Shared by several uses. Generally, it contains a parameter provided by the application program that is processed by the requested function. The API also can use this field to return a parameter to the application program. The alias names NEWEP, ASCB, and COUNT are used to reflect the function-specific use of this field.

PARM3 Shared by several uses. Generally, it contains a parameter returned by the API to the application program on completion of particular functions. In some cases, this field is used to pass additional information provided by the application program to the API. The alias names USER, DISCD, and DGERR are used to reflect the function-specific use of this field.

Variable-Length Parameters

Variable-length parameters follow the fixed-length parameters and are passed by reference. In this case, each variable-length parameter is identified by its address and length, stored within the TPL.

The following diagrams shows variable-length parameter format.



Variable-length parameters must be contained in the same address space as the TPL that references them.

Three variable-length parameters can be provided. They consist of:

- A protocol address
- User data,
- Protocol options

The area identified by one of these parameters can contain information supplied by the application program, information returned by the API, or both.

This list describes the TPL fields identifying variable-length parameters.

ADBUF Contains the address of a storage area used to pass a protocol address from the application program to the API or from the API to the application program. The relevant length is stored in the ADLEN field.

ADLEN Contains the length of a storage area whose address is stored in the ADBUF field. This field is used to define the length of protocol address information supplied by the application program or returned by the API.

DABUF Contains the address of a storage area used to pass arbitrary user data from the application program to the API or from the API to the application program. The relevant length is stored in the DALEN field.

DALEN Contains the length of a storage area whose address is stored in the DABUF field. This field is used to define the length of user data supplied by the application program or returned by the API.

- OPBUF Contains the address of a storage area used to pass protocol options from the application program to the API or from the API to the application program. The relevant length is stored in the OPLEN field.
- OPLEN Contains the length of a storage area whose address is stored in the OPBUF field. This field is used to define the length of protocol options supplied by the application program or returned by the API.

TPL Suffix

TPL suffix parameters allow the use of extended addresses. These suffix parameters append to the long form TPL and may contain ALETs of ESA extended addresses.

The TPL suffix is optional. An ALET (Access List Entry Token) is a value used in IBM Extended Addresses to designate the address space containing the referenced data area.

- PRM1X TPL extension for PARM1.
- PRM2X TPL extension for PARM2.
- PRM3X TPL extension for PARM3.
- ADALT An ALET corresponding to ADBUF.
- DAALT An ALET corresponding to DABUF.
- OPALT An ALET corresponding to OPBUF.
- XDIAG Extended diagnostic code for TPL.

The code includes a two-byte module identifier and a two-byte instance identifier.

Only fields that are referenced by a particular function need to be initialized. Also, many fields are optional and can be set to zero. The proper method to indicate that a variable-length parameter is missing is to set its length to zero.

Note: Setting a parameter's address to zero and specifying a non-zero value for its length is expressly prohibited and generates an error.

Using the in-line form of macro instructions frees you from having to allocate and manage TPLs, but at the expense of using more memory and being nonreentrant.

When the application program chooses to generate TPLs directly, two strategies suggest themselves:

- To generate a single TPL and reuse it for all requests
- To generate a TPL for each type of request and reuse it only for functions of the same type

The organization of the application program determines the best approach. The TPL provides a convenient mechanism for exchanging information with the API and the transport provider. It associates the information required for a single request into a self-contained unit that is more efficiently processed by the API. Since much of this information must be forwarded to the transport provider anyway, overhead is diminished by initially providing it in parameter list form; and because many transport functions require the same information, the same TPL can be reused with minimal effort.

Establishing a Session with a Transport Provider

Before the application program can begin issuing transport service requests, it must establish a session with the API. The term *session* is used loosely here to describe the infrastructure built and maintained by the API to service the application program. Establishing this session involves these processes:

- Locating the API subsystem
- Identifying the transport user
- Allocating necessary resources
- Initializing the interface environment

Session-Level Services

When the application program no longer requires the transport interface, the session should be terminated.

This table lists the session-level services provided by the API.

Function	Parameters	M/O	Description
AOPEN	APCB Address	M	Establishes session with the API and defines the transport user.
ACLOSE	APCB Address	M	Terminates session with the API.

Note: The column labeled M/O indicates whether a parameter is mandatory (M) or optional (O).

Application Program Control Block (APCB)

The API uses a data structure supplied by the application program as the primary anchor for information required to execute subsequent requests. This data structure is called an APCB.

Note: In many respects, the APCB is analogous to the ACB used by VTAM. A declarative macro instruction (also named APCB) can be used by the application program to generate this data structure.

The AOPEN and ACLOSE macro instructions have as their only operand the address of an APCB. The content of the APCB at the time it is opened determines the characteristics of the session. Some fields of the APCB are filled in by AOPEN and returned to their pre-opened state by ACLOSE. Other fields contain values supplied by the application program that define the parameters of the session.

An important parameter of the APCB is the MVS subsystem name for the API. If not specified by the application program, a default value is used. Otherwise, this must be the four-character ID of the API subsystem that is to service all future requests. Normally the default value suffices, but in those cases where more than one instance of the API is running, or where the name was changed during installation, this parameter must be specified in order to locate the correct subsystem.

AOPEN and ACLOSE Macros

A session is established with the API by issuing an AOPEN macro instruction that specifies the APCB to use for the session. This is called opening the APCB. All subsequent requests issued to the API must directly or indirectly reference an opened APCB. The session is terminated by closing the APCB with an ACLOSE macro instruction. Any resources allocated to the application program are released, and all endpoints associated with the APCB are closed.

The AOPEN Macro

This macro example shows how to establish a session with the transport provider.

```
*****
* ESTABLISH SESSION WITH API USING DEFAULT
* SUBSYSTEM
*****
TUNIT  AOPEN  TUAPCB  OPEN APCB FOR THIS TASK
        LTR    15,15  SESSION ESTABLISHED?
        BNZ    AOPENERR  IF NOT, GO TO ERROR ROUTINE
        .
        . {body of application program}
        .
TUAPCB  APCB    AM=TLI,APPLID=EXAMPLE DEFINE TRANSPORT USER
```

The ACLOSE Macro

This is an example of a macro for terminating a session with the transport provider.

```
*****
* TERMINATE SESSION WITH API BY CLOSING APCB
*****
TUTERM  ACLOSE  TUAPCB  CLOSE APCB FOR THIS TASK
        .
        . [no more service requests can be issued using this APCB]
        .
TUAPCB  APCB      AM=TLI,APPLID=EXAMPLE DEFINE TRANSPORT USER
```

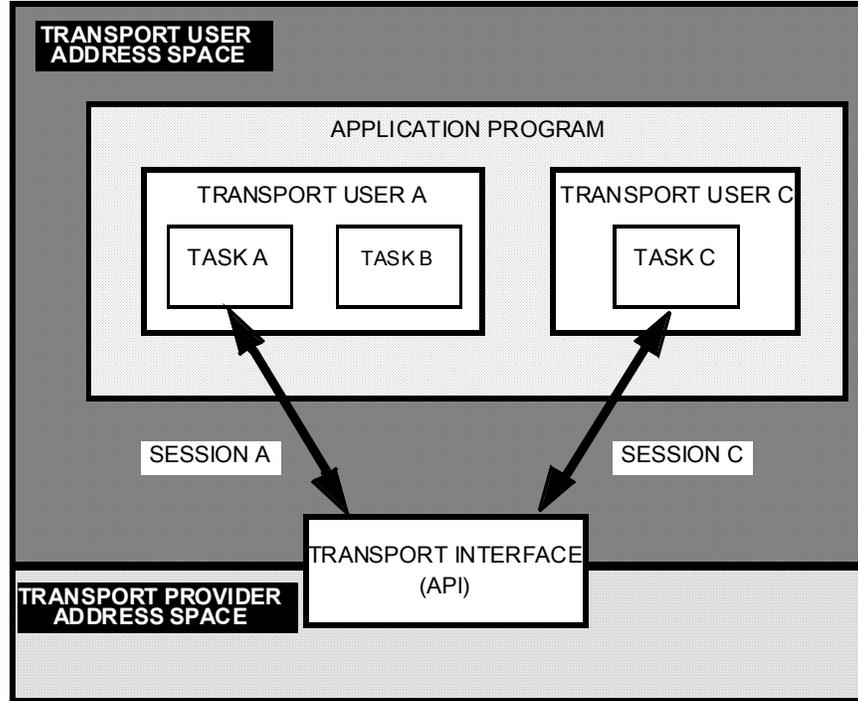
Application Programs and Transport Users

Until now, this guide referred to the application program and the transport user as if they were the same. Often they are, but there is a formal definition of the transport:

- The term *application program* is used throughout the remainder of this guide to refer, in general, to the program running in an address space using the API.
- The term *transport user* specifically refers to the task that opened the APCB. Since more than one task can execute in an address space, and since each task can open an APCB, an address space might have more than one transport user.

Application Program Example

This diagram shows an application program that consists of tasks A, B, and C.



Transport user A (represented by task A) and transport user C (represented by task C) has each opened an APCB and established a session with the API. Task B, a subtask of A, has not opened an APCB and cannot open any endpoints of its own. However, task B can use the infrastructure of task A to access endpoints opened by A.

The APCB gives an identity to the transport user and defines the context of its operation. Information associated with the APCB applies to the transport user in general, and indirectly to all of its endpoints.

Example

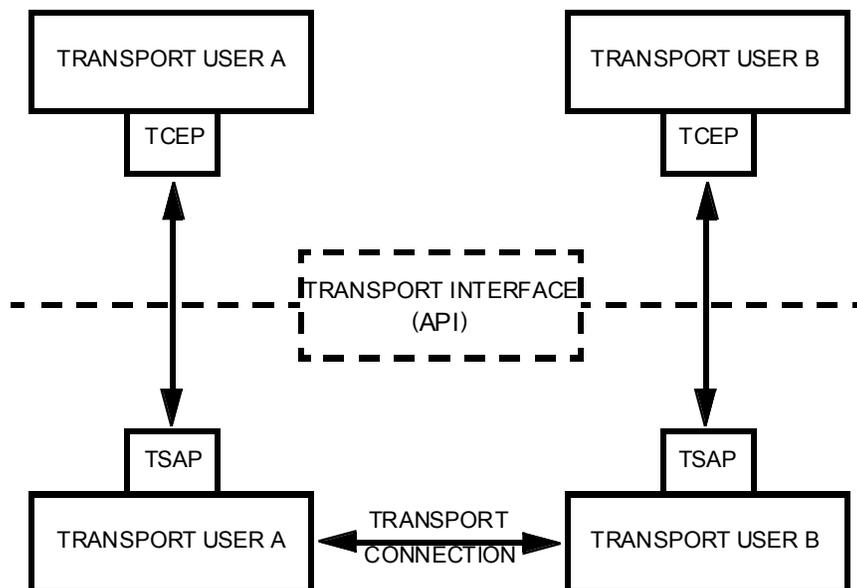
Exit routines that perform error recovery or process event indications are enabled via an exit list specified in the APCB. These exits apply to all endpoints created by the transport user unless specifically changed for a given endpoint. You can specify an arbitrary word of user context with the APCB that is passed as a parameter to exit routines. This lets a reentrant exit routine derive the context of a given transport user.

Connection-Mode Service

Connection-mode service is most appropriate for networking applications where the interaction between a transport user and its peer lasts for an extended period. File transfer and remote logon to time-sharing services are typical examples of applications that are well suited for connection-mode service.

Transport Connection Using connection-mode service, a transport user establishes a connection to a remote transport user.

The following diagram shows this transport connection. The two transport users communicate with one another through the transport interface and transport provider at each end of the connection. Each end of the transport connection is identified by the address of the Transport Service Access Point (TSAP) through which service is obtained and a connection ID corresponding to an endpoint within the service access point. The transport providers maintain this connection for the entire duration of data transfer, however long that may be. The Transport Connection Endpoint (TCEP) represents the user's interface to the TSAP.



Connection-mode service is characterized by these phases:

- Local endpoint management
- Connection establishment
- Data transfer
- Connection release

These phases parallel the three service phases defined by the OSI Reference Model with an additional phase for local endpoint management to handle those functions beyond the scope of the transport provider. Local endpoint management can be subdivided into an initialization phase that occurs before connection establishment and a termination phase that occurs after connection release. The following sections describe each phase in detail.

Local Endpoint Management

Local endpoint management consists of those services that are transparent to the transport provider. That is, local endpoint services do not require any interaction between the API and the transport provider to execute requests; they are implemented entirely within the transport interface. These services are used primarily to define and manipulate local information associated with endpoints.

Local endpoint services are provided via these groups of functions:

- Functions to open and close endpoints
- Functions to bind and unbind protocol addresses
- Miscellaneous functions for managing information and options associated with endpoints

The following table lists alphabetically all of the functions comprising local endpoint management.

Note: In the following table, and those that follow, the column labeled **M/O** indicates whether a parameter is mandatory (M) or optional (O). Parameters that are returned (R) or updated (U) are also indicated.

Function	Parameters	M/O	Description
TADDR	Endpoint ID Protocol Address	M MR	Returns protocol address bound to endpoint, or address of peer transport user.
TBIND	Endpoint ID Protocol Address Queue Length	M OU MU	Binds protocol address to endpoint and enables endpoint for receiving connect indications.
TCLOSE	Endpoint ID TCB Address ASCB Address	M O O	Closes endpoint, or transfers control to another task or address space.
TINFO	Endpoint ID User Data Address	M MR	Returns protocol information or statistics associated with endpoint.

Function	Parameters	M/O	Description
TOPEN	Domain	M	Establishes a new endpoint associated with a transport provider, or acquires control of an endpoint opened by another task or address space.
	Service Type	M	
	APCB Address	M	
	New Endpoint ID	MR	
	Old Endpoint ID	OU	
	Exit List Address	O	
	User Context	O	
	User ID	O	
	TCB Address	O	
	ASCB Address	O	
	Data Transfer Mode	O	
TOPTION	Endpoint ID	M	Negotiates protocol options associated with endpoint.
	Protocol Options	MU	
TUNBIND	Endpoint ID	M	Unbinds protocol address from endpoint, and disables endpoint from receiving connect indications.
TUSER	Endpoint ID	M	Associates a user-ID with endpoint for accounting and authorization.
	User ID	M	

Example

MR Indicates that the parameter is always returned.

OR Indicates that the parameter is returned only when the facility is supported by the transport provider, and a storage area has been provided by the transport user.

MU and **OU** Indicate that a parameter provided by the transport user can be updated by the transport provider.

Opening and Closing Endpoints

A transport endpoint is opened by executing the TOPEN function. If the request is valid and the appropriate resources are available, an endpoint is created in the indicated communications domain. On completion of the function, an identifier is returned that must be provided in all subsequent requests that reference the endpoint. The identifier is returned in the symbolic location TPLEP (the symbol TPLEPID can also be used) and is an unsigned fullword value. The application program should make no assumptions with regard to the format and content of the endpoint ID other than what has already been stated.

Opening an Endpoint

Only tasks that opened an APCB can open endpoints, and the address of the APCB must be included as a parameter of TOPEN.

This parameter permanently associates the endpoint with the transport user, and if the transport user terminates or closes the APCB, the endpoint is closed. The APCB also serves to identify the subsystem that services the endpoint

Note: TOPEN and AOPEN are the only functions that require an APCB parameter. All other functions locate the APCB via the endpoint ID.

The TOPEN Macro

The following is an example of a macro for establishing a session with the transport provider.

```
*****
* OPEN CONNECTION-MODE ENDPOINT USING TCP PROTOCOL
*****
EPINIT  TOPEN  DOMAIN=INET,TYPE=(COTS,ORDREL),APCB=TUAPCB
*
      LTR    15,15          OPEN ENDPOINT
      BNZ    TOPENERR      ENDPOINT CREATED?
      USING  TPL,1         IF NOT, GO TO ERROR ROUTINE
      L      9,TPLEPID     LOAD NEW ENDPOINT ID
      DROP   1
      .
      . [new endpoint ID can be used in subsequent requests]
      .
      TDSECT TPL          GENERATE TPL DSECT
```

where:

- DOMAIN Identifies the communications domain
- INET Specifies the Internet domain.
- TYPE Selects the mode of service.
- COTS Specifies connection-mode service.
- ORDREL Must be included as an optional sublist parameter of the service type if orderly release is required.

The DOMAIN and TYPE parameters are sufficient to select the transport provider and protocol that provides the service. Alternatively, a protocol number or service ID can be specified to make the selection in rare cases where the TYPE and DOMAIN parameters are ambiguous or to override installation defaults (for example, when multiple instances of the same provider exist).

A user ID can be associated with the endpoint. This information is used for authorizing access to services and accounting for their use. The particulars of how this information is used are not specified at this time, but it is anticipated that the user ID will be used to acquire access privileges from the local security system and will be included in any SMF data recorded by the API.

After an endpoint is opened, other API transport functions can be executed at the endpoint by supplying the endpoint ID with each function request. Therefore, the endpoint ID returned by TOPEN must be copied into any TPL used with subsequent service requests. This is usually done automatically by specifying the endpoint ID as an operand of the corresponding macro instructions. If the proper endpoint ID is already stored in the TPL, perhaps by a previous macro instruction, it is not necessary to code this operand.

Note: Failure to provide a valid endpoint ID with any API request (other than TOPEN OPTCD = NEW) causes the request to be rejected.

Defining Protocol
Event Notification for
an Endpoint

If the transport user requires asynchronous notification of certain protocol events, such as data arriving on a connection, the address of the exit routine or ECB should be included in an event notification list and indicated to the TOPEN function. Declarative macro instructions, TEXTLST and TEVNTLST, are provided to generate an event list.

The following protocol events are defined and correspond to particular service primitives issued by the transport provider.

CONNECT	Connect indication received.
CONFIRM	Connect confirmation received.
DATA	Normal data received.
XDATA	Expedited data received.
DISCONN	Disconnect indication received.
RELEASE	Release indication received.
SWIND	Send Window opened.

Exit Routines

Each event may have a different exit routine, or no exit routine at all. If no exit routine is specified for the event, or no exit list is specified for the endpoint, exit routines associated with the APCB are scheduled to handle these events. Refer to the chapter "Program Synchronization and Control" for a detailed discussion of asynchronous exit routines.

When exit routines are entered, the application program may need to derive some context associated with the endpoint to which the protocol event applies. Therefore, the API passes the endpoint ID as a parameter to the exit routine along with a word of user-defined context associated with the endpoint. This context word is specified when the endpoint is opened.

Event Control Blocks (ECBs)

If the transport user prefers, notification of certain protocol events can be requested using ECBs rather than an exit. An ECB is an MVS control block that is used to communicate between MVS services and application or system modules.

The primary difference between ECBs and exit routines is that exit routines are automatically scheduled when the requested operation completes, thereby saving the application program the trouble of waiting on and testing ECBs. On the other hand, the use of ECBs provides the program with greater control over the order in which events are handled.

The TEVNTLST macro was added to support protocol event notification ECBs. TEVNTLST supports ECBs and protocol event exits. TEXTLST supports exits only. TOPEN uses the EVENTLST parameter to support event lists consisting of exits and ECBs. EVENTLST is not supported by the APCB macro.

Fast-Path Authorized Exits

An authorized user can specify OPTCD=AUTHEXIT on the APCB macro or on the TOPEN macro. In this case, protocol and completion event notification exits are given control from an SRB rather than an IRB for the exit to run.

Closing an Endpoint

When the transport user is finished with an endpoint, it should be closed by executing a TCLOSE function.

Closing an endpoint causes any established connection to be released and any resources held by the endpoint to be relinquished.

The following is an example.

If a protocol address was bound to the endpoint, the address is unbound and made available for other endpoints.

Normally, connection release and address unbinding is done explicitly with separate service requests, but if the transport user needs to clean up immediately, a single TCLOSE is sufficient.

The TCLOSE Macro

The following is an example of a macro for terminating a session with the transport provider:

```
*****
* CLOSE ENDPOINT WHEN NO LONGER IN USE
*****
EPTERM  TCLOSE  EP=(9)  CLOSE AND DELETE ENDPOINT
      .
      . [endpoint must not be referenced after closing]
      .
```

Passing Control of an Endpoint

It may be convenient for the organization of the application program to have one task open an endpoint and another task close it. Therefore, provision was made to pass control of an endpoint from one task to another as long as each has opened an APCB. This capability also extends to pass control of an endpoint to a task in another address space.

To pass control, the current owner of the endpoint invokes the TCLOSE function specifying the ASCB and TCB address of the task that is to receive control of the endpoint. Similarly, the new task invokes TOPEN specifying the old endpoint ID as well as the ASCB and TCB addresses of the task passing control. An ASCB address of zero implies the current address space, and a TCB address of zero indicates that any task in the address space can pass or receive control of the endpoint.

A Typical Scenario

A typical scenario is for the receiving task to be a subtask of the controlling task and to have acquired the endpoint ID as one of its attach parameters.

Control is passed when each task rendezvouses at the same endpoint, one closing and the other opening. From the perspective of the old transport user, the endpoint is closed. From the perspective of the new transport user, a new endpoint is opened. However, the new endpoint has the same characteristics as the old endpoint, including the preservation of any connection that was established. The endpoint ID is changed and the old endpoint ID can no longer be referenced.

```
*****
* PASS ENDPOINT TO ANOTHER TASK IN THIS ADDRESS SPACE
*****
TU1PASS  ST      9,OLDEPID          STORE OLD ENDPOINT ID
      .
      . [task-1 attaches task-2 & passes endpoint ID in parmlist]
      .
      TCLOSE EP=(9),OPTCD=PASS  CLOSE AND PASS ENDPOINT
      LTR   15,15              ENDPOINT PASSED?
      BNZ   TU1FAIL           IF NOT, GO TO ERROR ROUTINE
      .
      . [code executed by task-2 follows]
      .
*****
```

```

* RECEIVE ENDPOINT FROM TASK WHICH ATTACHED THIS SUBTASK
*****
TU2PASS L    1,0(,1)          GET ADDRESS OF ENDPOINT ID
        L    9,0(,1)          LOAD OLD ENDPOINT ID
        AOPEN TU2APCB        OPEN APCB FOR THIS TASK
        LTR   15,15          SESSION ESTABLISHED?
        BNZ   TU2FAIL        IF NOT, GO TO ERROR ROUTINE
        TOPEN EP=(9),APCB=TU2APCB,OPTCD=OLD PASS OLD ENDPOINT
        LTR   15,15          ENDPOINT RECEIVED?
        BNZ   TU2FAIL        IF NOT, GO TO ERROR ROUTINE
        USING TPL,1
        L    9,TPLEPID        LOAD NEW ENDPOINT ID
        DROP 1
        .
        . [new endpoint ID is used in subsequent requests]
        .
*****
* CLOSE ENDPOINT WHEN NO LONGER IN USE
*****
TU2TERM TCLOSE EP=(9),OPTCD=DELETE CLOSE AND DELETE ENDPOINT
        .
        .
TU2PARM DC AL1(128),AL3(OLDEPID) ATTACH PARAMETER LIST
OLDEPID DS F                    OLD ENDPOINT ID
TU1APCB APCB AM=TLI,APPLID=TASK1 APCB FOR TASK-1
TU2APCB APCB AM=TLI,APPLID=TASK2 APCB FOR TASK-2
        TDSECT TPL            GENERATE TPL DSECT

```

Binding and Unbinding Addresses

Before any interaction with the transport provider can commence, a transport service access point must be assigned and associated with the endpoint. The address of this access point is the identifier that a peer transport user uses to connect to the local transport user. The TBIND function is used to assign the transport address.

The address of the access point can be assigned in one of two ways.

- If the transport user is a server, or expects the peer transport user to initiate the connection, it must specify the transport address using the protocol address parameter (ADBUF/ADLEN) of the TPL.
- If the transport user is a client, the transport address is specified as null and the transport provider assigns an unused transport address.

In either case, the communications domain identifies the transport address as the Internet domain and the network address portion of the protocol address is generally specified as a null address.

Whether or not the transport user intends to receive connect indications is also specified with the TBIND function. The transport user declares its intentions by specifying the size of the queue that holds pending connect indications. This queue is called the listen queue and is the source of information supplied to a transport user listening for incoming connection requests.

The size of the listen queue is specified with the QLSTN parameter. A value of zero indicates that no connect indications can be queued, and the endpoint is said to be disabled. A transport user operating as a client must use a disabled endpoint. A transport user operating in server mode must enable the endpoint by specifying a queue size greater than zero.

Using TBIND to Bind a Well-Known Address to a Server Endpoint

The OPTCD=USE option code is used to instruct the TBIND function to use the transport address provided by the user. This is generally the address of a well-known service, such as FTP, that is known to the peer transport user in advance. If the address is available, that is, not being used by another transport user and the requesting user has authority to use it, it is permanently associated with the endpoint, and this identifies the access point for all future services.

The depth of the listen queue (QLSTN = 5 in the following sample code) determines how many connect indications can be held at one time. However, it does not restrict how many clients can be simultaneously connected through the same access point. If the server is quick to respond to connect indications, or does not expect many simultaneous connection attempts, the size of the listen queue may be small. On the other hand, if the server is slow to accept connections, or anticipates that more than one transport user might be trying to connect at the same time, a higher value may be required. Except in special cases where a value of one is recommended (for example, a single-threaded server), usually a value of five is sufficient, even for servers supporting many connections.

```
*****
* BIND A WELL-KNOWN ADDRESS TO SERVER ENDPOINT
*****
SERVER  TBIND EP=(9),ADBUF=SERVERPA,ADLEN=LTPAINET,QLSTN=5,          +
        OPTCD=USE          BIND AND ENABLE ENDPOINT
        LTR 15,15          BIND SUCCESSFUL?
        BNZ TBINDERR      IF NOT, GO TO ERROR ROUTINE

        . [server can now listen for connect indications]
        .
SERVERPA DC AL2(TDINET),AL2(21),AL4(0) SERVER PROTOCOL ADDRESS
        TDSECT TPL,TPA,DOMAIN=INET  GENERATE INET TPA DSECT
* NOTE: LTPAINET and TDINET are defined in the
*       TPA and TPL macro expansions, respectively
```

Using TBIND to Bind Any Available Address to the Client Endpoint

If the transport user is a client, or intends to initiate the connection to the peer transport user, it does not require a specific transport address. Therefore, you can assign any address from a pool of available addresses. In this case, OPTCD=ASSIGN should be indicated with the TBIND function and the API assigns the transport address. A storage area for returning the assigned address can be specified with the protocol address parameter. If no storage area is provided, the transport address is assigned, but not returned to the transport user.

```

*****
* BIND ANY AVAILABLE ADDRESS TO CLIENT ENDPOINT
*****
CLIENT  TBIND EP=(9),ADBUF=CLIENTPA,ADLEN=L'CLIENTPA,QLSTN=0,      +
          OPTCD=ASSIGN          ASSIGN AND RETURN ADDRESS
          LTR 15,15              BIND SUCCESSFUL?
          BNZ TBINDERR           IF NOT, GO TO ERROR ROUTINE
          .
          . [client can now initiate connection to server]
          .
CLIENTPA DS XL(LTPAINET) CLIENT PROTOCOL ADDRESS
          TDSECT TPA,DOMAIN=INET GENERATE INET TPA DSECT

```

Binding and Listen Queue Relationship

Although the binding of a transport address and the allocation of a listen queue are two independent subfunctions of TBIND, they are related by the operating mode of the transport user. A client generally lets the API assign the transport address and specifies a queue size of zero. The server, on the other hand, generally binds a specific transport address of its choosing and enables the endpoint by specifying a queue size greater than zero.

These subfunctions can also be executed as two separate requests:

- An address can be bound with a QLSTN value of zero, leaving the endpoint disabled
- A second TBIND function can be executed later enabling the endpoint with a non-zero value for QLSTN

However, once an endpoint is enabled, the size of its listen queue cannot be changed. The endpoint should not be enabled until the server is ready to receive connect indications.

A server can use the listen queue to prioritize incoming connection requests. Since connect indications do not need to be accepted in the order they are presented, the server can gather several connect indications and accept them in priority order based on source address and protocol options (for example, quality of service). The depth of the queue represents the maximum number of indications that can be pending to the server.

A server that establishes multiple connections through the same access point requires an equal number of endpoints bound to the same transport address. Although it is valid (and necessary) to bind more than one endpoint to the same transport address, it is not valid to bind the same endpoint to more than one transport address at one time.

Using TUNBIND to
Unbind an Endpoint

Another transport address can be bound to an endpoint only after the previous address is unbound. This is done with the TUNBIND function.

The unbind service lets an endpoint be reused without closing and reopening. When a transport address is unbound, it becomes available for other transport users to use. If the endpoint was enabled, it can no longer queue connect indications. Closing an endpoint causes any bound transport address to be unbound as if the TUNBIND function had been issued.

```
*****
* UNBIND PROTOCOL ADDRESS AND DISABLE ENDPOINT
*****
UNBIND TUNBIND EP=(9) UNBIND PROTOCOL ADDRESS
      LTR 15,15 UNBIND SUCCESSFUL?
      BNZ TUNBNDERR IF NOT, GO TO ERROR ROUTINE
      .
      . [endpoint should be closed or reused with new address]
      .
```

Retrieving Protocol Addresses

A function related to TBIND and TUNBIND is TADDR. The TADDR service is used to retrieve protocol addresses associated with an endpoint.

Endpoints that are connected to a peer transport user are associated with two addresses:

- The local protocol address is the address bound to the endpoint by the TBIND function.
The local protocol address can be retrieved at any time after a successful TBIND is issued and is requested by indicating OPTCD=LOCAL with the TADDR function.
- The remote protocol address is the address of the connected peer transport user.

The remote protocol address can be retrieved only after a connection has been established and is indicated by OPTCD=REMOTE.

```
*****
* GET FULLY-QUALIFIED LOCAL PROTOCOL ADDRESS OF ENDPOINT
*****
GETADDR TADDR EP=(9),ADBUF=LOCALPA,ADLEN=L'LOCALPA, +
      OPTCD=LOCAL GET LOCAL PROTOCOL ADDRESS
      LTR 15,15 ADDRESS RETURNED?
      BNZ TADDRERR IF NOT, GO TO ERROR ROUTINE
      .
      . [if endpoint is connected, network address is returned]
      .
LOCALPA DS XL(LTPAINET) AREA FOR RETURNING PROTOCOL ADDR
      TDSECT TPA,DOMAIN=INET GENERATE INET TPA DSECT
```

TADDR is of questionable value since most of the information returned can be acquired through other means. However, the local protocol address returned by TADDR after a connection is established contains the network address, as well as the transport address and domain. Since multi-homed hosts cannot determine the local network address until the destination address is known, such information can only be retrieved in this manner. Nevertheless, knowing the local network address is generally not a requirement for most networking applications.

Miscellaneous Functions

The API service functions previously described must be invoked during the initial or final phase of service for every endpoint. The endpoint management functions described in this section are optional and generally can be invoked during other phases of service. These are the services in this category:

- Returning various information maintained by the transport provider
- Manipulating protocol options
- Specifying or changing the user ID and associated access privileges

The API and the transport provider maintain a variety of information associated with each endpoint, some of which may be of direct interest to the transport user. This information includes parameters and variables maintained by the transport provider that characterize the underlying protocol and the type of service available to the transport user, variables and control information that govern the protocol exchanges between transport layer entities, and statistical information that gives an accounting of the services provided. Some of this information is provider and protocol specific, while other information can be formatted and presented in a standardized fashion. The discussion in this section focuses on that information common to all transport providers.

TINFO—Getting Basic Protocol Information

Information is obtained using the TINFO service function. The user data parameter identifies a storage area provided by the application program for returning information, and an option code specifies the type of information desired. The TINFO service is the only API function that uses the user data parameter for returning information not received from the peer transport user.

```
*****
* OBTAIN BASIC PROTOCOL INFORMATION ABOUT TRANSPORT PROVIDER
*****
GETINFO LA      8,INFOAREA      LOAD ADDRESS OF DATA AREA
        TINFO  EP=(9),DABUF=(8),DALEN=L'INFOAREA,      +
        OPTCD=PRIMARY      GET PROTOCOL INFORMATION
        LTR    15,15      INFORMATION RETURNED?
        BNZ    TINFOERR      IF NOT, GO TO ERROR ROUTINE
        USING  TIB,8
        .
        . [information returned can be used for run-time configuration]
        .
INFOAREA DS      XL(TIBLEN)      AREA FOR RETURNING TIB
        TDSECT TIB      GENERATE TIB DSECT
```

Basic protocol information that has been standardized for all transport providers is requested by indicating OPTCD=PRIMARY. The information returned can be used by the application program to determine basic characteristics of the transport service (for example., maximum lengths of protocol addresses, user data, and protocol options are provided). Whether or not certain facilities are supported can also be determined. The intent of this information is to let the application program interpret it at runtime and thereby adapt to the specific characteristics of the transport service. A program that correctly applies this information should be readily portable from one transport provider to another.

The Transport Information Control Block (TIB)

This information database is returned as a fixed-length data structure called a TIB. The TIB is mapped by a DSECT generated by the TDSECT macro instruction and is listed in its entirety in *TCPAccess Assembler API Macro Reference*.

This list summarizes the type of information available:

- The communications domain and mode of service requested
- Basic characteristics and options of the underlying protocol
- The MVS subsystem name, the API service name, and protocol number
- Limits associated with various interface services
- Limits associated with various provider services

The *TCPAccess Assembler API Macro Reference* gives a detailed description for the TINFO macro and the basic protocol information returned by the TINFO service.

TOPTION—Manipulating Protocol Options

Protocol options are manipulated with the TOPTION service. The protocol options parameter identifies a storage area containing a list of options to be manipulated, and in some cases, a desired value for each option. An option code (that is, the OPTCD parameter) provided with the request indicates the action the transport provider should take.

These services are available:

- DEFAULT Returns the default values of the indicated options.
- QUERY Returns the current values of the indicated options.
- VERIFY Verifies whether the value indicated for each option is supported, and if not, returns the superior value supported.
- DECLARE Sets the specified options to the indicated values and returns any options negotiated to an inferior value.

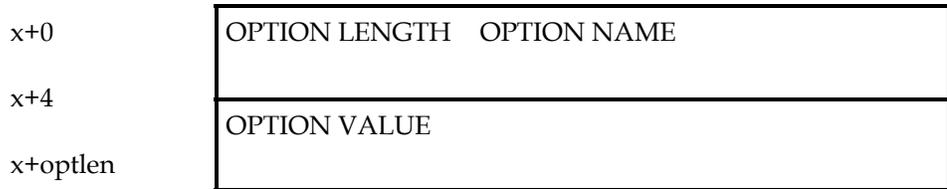
Option codes (OPTCD) also indicate whether the options to be manipulated are transport provider (TP) or interface options (API). TP provider options are provider-specific and protocol-dependent. Transport interface options are independent of any particular provider and only affect facilities within the API.

```
*****
* DECLARE TRANSPORT INTERFACE OPTIONS
*****
SETOPTN TOPTION EP=(9),OPBUF=APIOPTN,OPLN=LENOPTN,
          OPTCD=(DECLARE,API) DECLARE API OPTIONS
          LTR 15,15 OPTIONS ACCEPTED?
          BNZT TOPTNERR IF NOT, GO TO ERROR ROUTINE
          .
          .
APIOPTN DC AL2(8),AL2(TPOAQSND),AL4(4) MAX NO. OF SEND REQS
          DC AL2(8),AL2(TPOAQRCV),AL4(1) MAX NO. OF RECV REQS
          DC AL2(8),AL2(TPOALSND),AL4(65536) LEN OF SEND BUFFER
          DC AL2(8),AL2(TPOALRCV),AL4(4096) LEN OF RECV BUFFER
LENOPTN EQU *-APIOPTN
          TDSECT TPO GENERATE TPO DSECT
```

Protocol Options List Format

Although the number, type, and value of protocol options can vary from one provider to the next, a common structure is used to format the protocol options list.

This list is variable in length and consists of an arbitrary number of option entries, each of which is formatted as shown below.



Protocol Option Entries:

- OPTION LENGTH The total length of the option entry.
- OPTION NAME The name of the option.
- OPTION VALUE The actual value of the option.

The transport provider can impose some additional structure on the option name field (for example, the option name can identify a protocol level in addition to an option number). This follows from the observation that with some protocol stacks (for example, Internet), the transport user might want to set network-level options such as source routes, as well as transport-level options. The basic structure of an option entry is defined by the TPO DSECT, which also defines the option names for API options.

When using the QUERY and DEFAULT forms of TOPTION, the transport user should build an options list initializing the length and name field of each option entry. On completion of the TOPTION function, the value of each option is returned in the value field. When using the VERIFY and DECLARE forms, all three fields should be initialized. On completion of the request, any value that was invalid and negotiated is updated in place.

TUSER—Specifying or Changing an Endpoint User ID

The last service function in the miscellaneous group is TUSER. This function is used to specify or change the user ID associated with an endpoint. If a user ID is not specified when the endpoint is opened, it can be specified later with the TUSER function. In interactive applications serving multiple users, the user ID may not be known until the peer transport user has logged on. Therefore, the TUSER function can be invoked after the peer transport user connects, and even after data is exchanged.

The API uses two alternative structures for providing the user ID. The first is a simple API structure consisting of a user ID, group name, and password. This structure is called a transport endpoint user block and is mapped by the TUB DSECT generated by the TDSECT macro instruction.

```
*****
* SPECIFY OR CHANGE ENDPOINT USER ID AFTER TOPEN
*****
SETUSRID  TUSER EP=(9),USER=TUBAREA SET USER ID
          LTR 15,15                USER ID ACCEPTED?
          BNZ TUSERERR             IF NOT, GO TO ERROR ROUTINE
          .
          .
TUBAREA   DC XL(TUBLEN)'00'        TRANSPORT ENDPOINT USER BLOCK
          ORG TUBAREA+TUBUID-TUB
          DC AL1(7),CL8'CSS31J4'   USER ID
          ORG TUBAREA+TUBPWD-TUB
          DC AL1(8),CL8'ROSEWOOD'  PASSWORD
          ORG
          TDSECT TUBGENERATE       TUB DSECT
```

The second structure is an Accessor Environment Element (ACEE). The ACEE is an MVS data structure used by the resident security system for maintaining user and security information.

Presently, the API uses this information for informative and diagnostic purposes only. However, it is anticipated that at some future date the information provided will be authenticated with the resident security system, and access privileges associated with the user ID will be used to authorize access to networking facilities.

Connection Establishment

The connection establishment phase highlights the fundamental differences between client and server mode. The transport interface imposes a different set of procedures in this phase for each type of transport user. The client initiates connection establishment by requesting connection to a server at a particular destination address. The server, on the other hand, waits for connection requests and is notified via connect indications issued by the transport provider. The server can either accept or reject the client's request. If the request is accepted, the connect response issued to the transport provider causes the client to be notified with a connect confirmation. Otherwise, a disconnect indication is issued.

These connection procedures are implemented by six of the API service functions.

The following table lists the API service functions in alphabetical order.

Note: The column labeled **M/O** indicates whether a parameter is mandatory (M), optional (O) or returned (R).

Function	Parameters	M/O	Description
TACCEPT	Endpoint ID New Endpoint ID Sequence Number	M O M	Accepts connect indication and establishes connection to calling transport user using designated endpoint as the new end-point.
TCONFIRM	Endpoint ID Protocol Address	M OR	Confirms when a connection has been established to the called transport user.
TCONNECT	Endpoint ID Protocol Address	M M	Requests that a connection be established to the designated transport user.
TLISTEN	Endpoint ID Protocol Address Sequence Number	M OR MR	Listens for connect indications.
TREJECT	Endpoint ID Sequence Number	M M	Rejects connect indication from calling transport user and abandons connection establishment.
TRETRACT	Endpoint ID	M	Retracts an outstanding TLISTEN request.

TCONNECT—Initiating a Connection

The client initiates a connection request using the TCONNECT function. The endpoint must be opened, bound to a local transport address, and disabled. The destination protocol address must be provided with the request. The TCONNECT function completes as soon as the service primitive is issued to the transport provider. In particular, the successful completion of a TCONNECT request does not constitute the establishment of a connection.

```
*****
* INITIATE CONNECTION TO FTP SERVER AT 127.0.0.1
*****
CONNECT  TCONNECT EP=(9),ADBUF=SERVERPA,ADLEN=LTPAINET
*
          LTR  15,15          INITIATE CONNECT
          BNZ  TCONNERR      CONNECTION INITIATED?
                              IF NOT, GO TO ERROR ROUTINE
*****
* WAIT FOR CONNECTION ESTABLISHMENT TO BE CONFIRMED
*****
CONFIRM  TCONFIRM EP=(9)      WAIT FOR CONFIRMATION
          LTR  15,15          CONNECTION ESTABLISHED?
          BNZ  TCONFERR      IF NOT, GO TO ERROR ROUTINE
          .
          . [the endpoint is now ready for data transfer]
          .
SERVERPA DC  AL2(TDINET),AL2(21),AL1(127,0,0,1)
*
          TDSECT TPL,TPA,DOMAIN=INET  GENERATE TPL AND TPA DSECTS
* NOTE: LTPAINET and TDINET are defined in the
*       TPA and TPL macro expansions, respectively
```

The client must wait for confirmation. The client receives confirmation by issuing the TCONFIRM function. TCONFIRM does not complete until a connect confirm is issued by the transport provider, or if the request was rejected, until a disconnect indication is received. The protocol address of the destination is returned to the transport user. The destination protocol address should be the same as that provided to TCONNECT. If the client does not require the protocol address value, the corresponding parameters should be set to zero.

The client may issue the TCONFIRM function in anticipation of the server accepting the request or may wait for explicit notification that the confirmation was received. In the former case, and when operating in synchronous mode, the client is suspended until the server responds. In the latter case, an asynchronous exit routine is entered, giving notification that the confirmation arrived. The TCONFIRM function should then be executed. It completes immediately without suspending the issuing task. Use of exit routines is described separately in the chapter “Program Synchronization and Control.”

If the server rejects the connection request, or some malfunction in the network prevents establishment of the connection, a disconnect indication is issued by the transport provider. This causes any pending TCONFIRM functions to complete with an error, or if asynchronous exits are enabled, the disconnect exit routine is scheduled. Therefore, the client is always notified in the event of an unsuccessful connection attempt.

Single-Threaded and Multithreaded Servers

The procedures used by the server varies depending on its internal organization:

- Single-threaded servers generally service one connection at a time and are similar in complexity to clients
- Multithreaded servers are often more complex and can service many connections simultaneously

Single-Threaded Servers

The single-threaded server requires an opened endpoint bound to a well-known transport address. The endpoint must also be enabled, generally with a queue size of one. The subsequent operation of the server differs from the client in that the client initiates a connection request and then waits, whereas the server waits for a connection request and then responds. The mechanisms for synchronization parallel those of the client.

TLISTEN—Receiving a Connect Indication

The server receives connect indications using the TLISTEN function. When TLISTEN completes, the protocol address of the client is returned to the server in storage areas provided with the initial request. The server must use this information to determine whether it should connect to the client and accept or reject the request.

If the transport user anticipates the connection request and invokes the TLISTEN function in advance, the issuing task is suspended when operating in synchronous mode. The TLISTEN function subsequently completes when a connect indication has been received.

```
*****
* LISTEN FOR CONNECTION REQUESTS ARRIVING AT ENDPOINT
*****
LISTEN    TLISTEN EP=(9),ADBUF=CLIENTPA,ADLEN=L'CLIENTPA
*
          LTR    15,15                INITIATE LISTEN
          BNZ    TLSTNERR             CONNECT INDICATION RECEIVED?
          USING  TPL,1                IF NOT, GO TO ERROR ROUTINE
          L      7,TPLSEQNO           LOAD SEQUENCE NUMBER
          DROP   1
          .
          . [server determines whether to accept or reject]
          .
CLIENTPA  DS    XL(LTPAINET)          ADDRESS OF CLIENT
          TDSECT TPL,TPA,DOMAIN=INET  GENERATE TPL AND TPA DSECTS
```

Alternatively, the server can provide an exit routine to be scheduled when connect indications arrive. The TLISTEN function should then be issued within the exit routine and completes without suspension of the issuing task.

TACCEPT—Accepting a Connect Indication

A connect indication is accepted using the TACCEPT function. The responding protocol address should be the same as that bound to the endpoint and does not need to be provided to the TACCEPT function.

While the transport user is ruling on whether or not to accept the connection, the API must retain the connect indication in the endpoint's listen queue. A sequence number is returned with the completion of TLISTEN that uniquely identifies the entry in the queue. This sequence number must be provided with the corresponding TACCEPT request to identify the accepted indication. Even if the queue size is one and the intent is unambiguous, the transport user must always supply a valid sequence number.

When the TACCEPT function completes, a connection has been established between the client and server. The endpoint is now ready for data transfer and is unable to receive more connect indications until the connection is released. Any TLISTEN function issued to a connected endpoint completes with an error.

```
*****
* ACCEPT CONNECTION IN SINGLE-THREADED MODE
*****
STHREAD  TACCEPT EP=(9),SEQNO=(7)  ACCEPT CONNECTION REQUEST
        LTR  15,15                  CONNECTION ESTABLISHED?
        BNZ  TACPTERR                IF NOT, GO TO ERROR ROUTINE
        .
        . [the connection can now be used for data transfer]
        .
```

TREJECT—Rejecting a Connection Indication

A connect indication is rejected using the TREJECT function. A sequence number provided with the request identifies the rejected indication. TREJECT is actually a disconnect service.

Since connections are established by the transport provider independently from TU actions, a TREJECT causes the session to be abortively disconnected. Any data that was sent (in this case—from the client), is lost, even though it may have been acknowledged by the transport provider.

If the transport provider supports disconnect user data, user data may be provided by the server for sending to the client with the subsequent disconnect indication. Connection requests can be rejected for any number of reasons, and disconnect user data is a convenient method of advising the client as to why the request was rejected. Whether or not this facility is supported by the transport provider can be determined easily at runtime by examining the contents of the TIB.

```
*****
* REJECT REQUEST FOR CONNECTION
*****
REJECT   TREJECT EP=(9),SEQNO=(7)  REJECT CONNECTION REQUEST
        LTR  15,15                  CONNECTION ABANDONED?
        BZ   LISTEN                  IF SO, LISTEN FOR NEXT CLIENT
        .
        . [handle error condition on endpoint]
```

Multithreaded Servers

A multithreaded server cannot tie up the endpoint at which it expects to receive connection requests, and therefore must implement some additional steps.

The customary procedure is to create additional endpoints for establishing connections and to leave the original endpoint free to listen for connect indications. The number of additional endpoints the server is prepared to open determines the number of simultaneous connections it is able to service. The size of the listen queue only restricts the number of pending connect indications that may be awaiting acceptance or rejection.

```
*****
* ACCEPT CONNECTION IN MULTI-THREADED MODE
*****
MTHREAD  TOPEN DOMAIN=INET,TYPE=(COTS,ORDREL),APCB=TUAPCB
*
      LTR 15,15          OPEN ENDPOINT
      BNZ TOPENERR      ENDPOINT CREATED?
      USING TPL,1       IF NOT, GO TO ERROR ROUTINE
      L    6,TPLEPID     LOAD NEW ENDPOINT ID
      DROP 1
      TBIND EP=(6),ADBUF=SERVERPA,ADLEN=LTPAINET,QLSTN=0,
      OPTCD=USE          BIND AND LEAVE DISABLED
      LTR 15,15          BIND SUCCESSFUL?
      BNZ TBINDERR      IF NOT, GO TO ERROR ROUTINE
      TACCEPT EP=(9),SEQNO=(7),NEWEP=(6)
*
      LTR 15,15          ACCEPT TO NEW ENDPOINT
      BNZ TACPTERR      CONNECTION ESTABLISHED?
      IF NOT, GO TO ERROR ROUTINE
      . [the new endpoint is now ready for data transfer]
      .
SERVERPA DC AL2(TDINET),AL2(21),AL4(0)
*
      TDSECT TPL,TPA,DOMAIN=INET GENERATE INET TPA DSECT
* NOTE: LTPAINET and TDINET are defined in the
* TPA and TPL macro expansions, respectively
```

The Listening Endpoint

The listening endpoint is kept available for receiving connect indications by accepting the connection to a new endpoint. The new endpoint must be opened and disabled, then bound to the same well-known address.

The new endpoint must have been opened by the same task that opened the endpoint receiving the connect indication. The endpoint ID is provided as a parameter to TACCEPT, and the connection is established to the indicated endpoint.

If the value of the endpoint ID is zero, or identifies the listening endpoint, the connection is established to that endpoint as previously described for single-threaded servers. Otherwise, the connection is established to the new endpoint. When the TACCEPT function completes, the new endpoint is ready for data transfer and the old endpoint can continue to be used for receiving connect indications from other clients.

The server is not required to accept connect indications in the order received. If several connect indications are available at one time, the server may receive all of them, and using the appropriate sequence numbers, can accept or reject them in what ever order it chooses. This technique can be used to give priority to certain classes of clients, particularly if resources are limited (for example, endpoints).

The synchronization aspects of multithreaded mode and single-threaded mode are similar, except that multithreaded servers must be more careful about being suspended. This follows from the transport user needing to service many connections, which is not possible when it is suspended for long periods. The strategies that can be employed are to create a subtask for each connection and to pass control of the connected endpoint to the subtask, or to implement a dispatching loop to service individual endpoints and to use asynchronous execution modes to prevent indefinite suspension of the task.

TRETRACT—Retracting
a Previous Listen
Request

Once the TLISTEN function is invoked at an endpoint, it normally does not complete until a connection request arrives. If the transport user wants to discontinue listening for connect indications, it can either close the endpoint or retract the pending TLISTEN with a TRETRACT function. The latter has the advantage of being able to reissue the TLISTEN function without opening a new endpoint. The TRETRACT serves only to undo the effects of an uncompleted TLISTEN. If the TRETRACT function completes successfully, the state of the endpoint is as if the TLISTEN request had never been issued.

```
*****
* RETRACT PREVIOUS LISTEN REQUEST
*****
RETRACT   TRETRACT EP=(9)           RETRACT OUTSTANDING LISTEN
          LTR   15,15                LISTEN RETRACTED?
          BZ   NOLISTEN              IF SO, BRANCH AROUND
          CH   0,=AL2(TAEXCPTN)     EXCEPTIONAL CONDITION?
          BNE  TREERR                IF NOT, GO TO ERROR ROUTINE
          USING TPL,1
          CLI  TPLERRCD,TENOLSTN    LISTEN ALREADY COMPLETED?
          BNE  TREERR                IF NOT, GO HANDLE ERROR
          DROP 1
NOLISTEN  DS   OH
          .
          . [listen has been retracted or already completed]
          .
          TDSECT TPL                 GENERATE TPL DSECT
```

Data Transfer

The distinction between client and server becomes unimportant after a connection has been established. Each may send or receive data on the connection, and when data transfer is complete, either may initiate its release. During the data transfer phase, the API and the transport provider work together to ensure the reliable transfer of data between the transport user and its peer, without duplication or loss of data.

TLI vs. Sockets Mode

Once a connection is established, data transfer is performed in the data transfer mode specified on the TOPEN macro. The operation of these modes is defined in [Concepts and Terminology](#).

Connection-Oriented Transport Service (COTS) Data Transfer Functions

The following table summarizes the API service functions implemented for data transfer.

Function	Parameters	M/O	Description
TRECV	Endpoint ID	M	Receive data from peer transport user.
	User Data	MR	
	Residual Count	MR	
TSEND	Endpoint ID	M	Send data to peer transport user.
	User Data	M	
	Residual Count (TLI mode)	MR	
	Sent Data Count (Socket mode)	MR	

Transporting User Data

These types of user data can be transferred over a transport connection:

- Normal data is supported by all transport providers and is delivered to the destination transport user in the same order it was received from the source.
- Expedited data typically is associated with information of an urgent nature and may be delivered ahead of normal data. The exact semantics of expedited data are subject to the interpretations of the transport provider. Furthermore, not all transport protocols support the notion of expedited data. The TIB returned by TINFO may be examined at runtime to determine whether a transport provider supports the transfer of expedited data.

While a given transport provider may guarantee the integrity and order of user data, it does not guarantee that data transmitted as a single unit by one transport user is delivered to its peer as a single unit. Data units can be split or combined with previous or subsequent data units as long as the data is delivered in order and without duplication. The term *data unit* as used here generally refers to an arbitrary quantity of data transmitted by a transport user. If a transport connection is likened to a pipe between a source and sink of data, the aggregation of data as it enters the pipe may not be preserved as it exits.

This form of data transfer is called a byte stream. If the application program is careful not to make any assumptions about the physical aggregation of data and embeds the necessary information within the byte stream itself if message boundaries must be preserved, then the program should be capable of operating over the transport protocol without regard to such factors as network data loss and retransmission, fragmentation and reassembly, timing dependencies, or windowing constraints.

The unit of data exchanged between the transport user and the transport interface is called a Transport Interface Data Unit (TIDU). The unit of data exchanged between the transport user and the transport provider is called a Transport Service Data Unit (TSDU). The maximum size of a TIDU is interface-dependent, and the maximum size of a TSDU is provider-dependent. Both may be determined at runtime by examining the TIB returned by the TINFO service. The maximum size of a TSDU may be large (possibly unlimited), and in particular, can be larger than the maximum size of a TIDU. Therefore, the API lets a TSDU be transmitted or received as multiple TIDUs. Furthermore, the size of each TSDU is preserved as it is transferred over the connection, but may be received by the peer transport user as a different number of TIDUs (for example, the size of a TIDU is a local characteristic and the size of a TSDU is a global characteristic).

EOM/NOTEOM—
Delineating End of
Transport

The option code EOM/NOTEOM is used to delineate the end of a TSDU.

- When sending data, the transport user must assert NOTEOM if the TSDU will be continued with the next TIDU, or assert EOM if the TIDU contains the end of the TSDU. In the latter case, the last byte of the TIDU corresponds to the last byte of the TSDU.
- When receiving data, the API sets the option code and the transport user interprets it. That is, if the TSDU is continued with one or more TIDUs, NOTEOM is asserted when the receive function completes, and EOM is asserted when the end of the TSDU is received.

Option codes are asserted by the setting or clearing of flag bits in the OPTCD field of the TPL. NOTEOM is asserted when the corresponding flag bit is set, and EOM is asserted when the flag bit is not set.

OPTCD=EOM is the default. OPTCD=NOTEOM is ignored by the API if set by the transport user when TSDU message boundaries are not supported by the transport provider.

MORE/NOMORE—
Indicating Additional
Data

The option code MORE/NOMORE indicates whether or not there is more data available.

- For send functions, MORE indicates that the transport user has more data to send and expects to immediately issue another send request.
- When MORE is indicated at the end of a receive function, it indicates that more data is immediately available from the transport provider

MORE and NOMORE are unrelated to EOM and NOTEOM. That is, MORE indicates there is more data available, and NOTEOM indicates that the data is part of the same TSDU.

An indication of MORE on a send request advises the transport provider that a subsequent send request is assured and probably follows immediately. The transport provider may want to delay sending any partially filled protocol data unit hoping to append data from the subsequent send. On the other hand, an indication of NOMORE is a signal that no more data may follow, and any partially filled data unit should be transmitted immediately.

For endpoints using TCP as the transport protocol, an indication of NOMORE becomes a *PUSH* at the transport provider interface. An indication of MORE on the completion of a receive request serves only to signal the user that more data is immediately available for a subsequent receive request. If the request is issued in synchronous mode, the issuing task is not suspended.

NORMAL/EXPEDITE—
Indicating the Type of
Data

The option code NORMAL/EXPEDITE indicates the type of data. On a send request, NORMAL indicates that the user data should be sent as normal data. EXPEDITE indicates that the data should be sent as expedited data. Similarly, on the completion of a receive request, NORMAL and EXPEDITE indicate the type of data transferred.

The interpretation applied by the transport provider may vary. For example, an ISO TP provider actually transfers the data as an expedited data unit and delivers it to the receiver as a separate data unit, perhaps ahead of undelivered normal data. The maximum size of an expedited data unit can be determined by examining the TIB. TCP, on the other hand, does not transfer expedited data per se, but rather transmits an urgent condition and records the location in the data stream where the urgent condition occurred. The receiver is then signaled that the urgent condition exists, which persists until the recorded position in the data stream is reached.

The transport user does not request to receive expedited data; rather, the user is told on completion of a request whether or not expedited data was transferred. For the ISO transport user, the interpretation is quite simple: the data transferred is expedited data and should be handled in a manner appropriate for the application program. For the Internet transport user, the interpretation is quite different. In this case, the EXPEDITE option code indicates there is some urgent data in the data stream, and the transport user should process the current (normal) data as expeditiously as possible in order to receive the urgent data. In many cases, this may mean discarding data up to the point of urgent data. Urgent data has no actual length, and begins with the first data unit received without the EXPEDITE option code set.

Sending and Receiving Data

Data is sent with the TSEND function and received with the TRECVC function. The user data parameter identifies the storage area containing the data to be sent or the storage area in which the API returns the received data.

When a TRECVC request completes:

- The user data parameter is updated to reflect the actual amount of data received
- The transport user also must set the appropriate option codes for TSEND
- The API sets the option codes prior to the completion of TRECVC

The user data may be in direct or indirect format as described in [Transport User Data](#). The total amount of user data cannot exceed the limits defined for send and receive TIDUs. This information can be found in the TIB.

When a TSEND function is executed, the user data is immediately moved into an internal buffer allocated in the address space of the transport provider. This prevents the transport user's storage area from being tied up until the data is sent and allows swapping of the address space. Similarly, TRECVC reserves the appropriate amount of space in the transport provider's address space for receiving the requested data. The data can then be received from the transport provider while the transport user's address space is swapped out. On completion, the address space is swapped back in, and the data is moved into the transport user's storage area. The total amount of send and receive buffering that can be in use by an endpoint is limited. Default values are defined separately for send and receive at installation time and can be modified with the TOPTION service. The maximum permitted values are defined in the TIB.

```
*****
* SEND LOGON PROMPT AND RECEIVE REPLY
*****
LOGON  TSEND EP=(9),DABUF=PROMPT,DALEN=PROMPTLN,
      OPTCD=NOMORE          PROMPT FOR USER ID
      LTR 15,15             DATA SENT SUCCESSFULLY?
      BNZ TSENDERR          IF NOT, GO TO ERROR ROUTINE
      TRECVC EP=(9),DABUF=REPLY,DALEN=L'REPLY RECEIVE REPLY
      LTR 15,15             REPLY RECEIVED?
      BNZ TRECVCERR          IF NOT, GO TO ERROR ROUTINE
      USING TPL,1
      .
      .[parse reply data]
      .
      TM  TPLOPCD2,TOMORE    MORE DATA TO RECEIVE?
      BO  RECVMORE           IF SO,GO RECEIVE IT
      DROP 1
      .
      . [this is a contrived example]
      .
PROMPT DC  AL1(13),AL1(37)   NEW LINE
      DC  C'PLEASE LOG ON'   HERALD MESSAGE
      DC  AL1(13),AL1(37)   NEW LINE
      DC  AL1(13),AL1(37)   NEW LINE
      DC  C'ENTER USER ID: ' USER ID PROMPT
PROMPTLN EQU *-PROMPT
REPLY DS XL80              REPLY AREA
      TDSECT TPL            GENERATE TPL DSECT
```

The transport user can anticipate the arrival of data and issue a TRECVC request in advance.

- If no data is available and the request was issued in synchronous mode, then the issuing task is suspended until data is received
- If no data is available and the request was issued in asynchronous mode, the request remains pending and completes asynchronously when a data indication arrives

Alternatively, the transport user can wait until data is available, and then receive it without suspending the task. The latter method is enabled by specifying an exit routine when the endpoint (or APCB) is opened. Separate exit routines can be specified for normal or expedited data.

Unlike many of the other transport service functions, the API lets more than one TSEND or TRECVC be outstanding at any given time on any given endpoint. This is to allow a sufficient amount of overlap with transport provider processing and latency due to physical network I/O. Other functions must normally complete before another function can be issued at the same endpoint. The number of pending send and receive requests is limited, however. Like the preceding buffer values, the default send and receive limits are defined at installation time and can be modified with the TOPTION service. The maximum permitted values are defined in the TIB. The buffer size and pending request limits cannot be changed after the first TSEND or TRECVC function is issued.

User Data Length

The total length associated with the user data parameter is simply the user data length for a direct request or the sum of all segment lengths for an indirect request. This length must not exceed any one of these values:

- The maximum size of a transport interface data unit
- The maximum size of a transport service data unit
- The negotiated amount of internal buffer space minus the total accumulated length of all other pending requests

Often an application program uses fixed-length send and receive buffers in its own address space. In this case, the transport user can prevent overrunning API buffer space by limiting the number of buffers and choosing an appropriate value for the maximum number of pending requests.

Connection Release

At any point during data transfer, either user can release a transport connection. The transport provider also can release the connection as the result of some nonrecoverable network malfunction or protocol error.

The API supports these forms of connection release:

- An abortive release, where the connection is released immediately, and undelivered user data may be discarded
- An orderly release, where all previously sent user data is delivered to the transport user before the connection is released

All transport providers must support abortive release. Orderly release is optional and must be requested when the endpoint is opened.

The following table summarizes the API service functions for connection release.

Function	Parameters	M/O	Description
TCLEAR	Endpoint ID Sequence Number Reason Code	M MR MR	Clear (receive) pending disconnect indication.
TDISCONN	Endpoint ID	M	Release connection, or abandon connection establishment.
TRELACK	Endpoint ID	M	Acknowledge (receive) pending orderly release indication.
TRELEASE	Endpoint ID	M	Request the orderly release of a connection.

TDISCONN—Initiating Abortive Release

A transport user initiates abortive release by invoking the TDISCONN service function. This causes the connection to be released immediately, and a disconnect indication is delivered to the peer transport user. Any user data previously sent with a TSEND function that was not delivered to the destination transport user may be discarded.

```
*****
*  ABORT CONNECTION BY INITIATING DISCONNECT
*****
ABORT  TDISCONN EP=(9)          INVOKE DISCONNECT FUNCTION
        LTR      15,15          CONNECTION RELEASED?
        BNZ      TDISCERR      IF NOT, GO TO ERROR ROUTIN
        .
        . [connection is released, no data transfer is allowed]
        .
```

The API learns of an abortive release by receiving a disconnect indication from the transport provider. The transport user is notified of this occurrence either by terminating the next service request with an error indicating the disconnect or by scheduling the user's exit routine, if one has been enabled.

TCLEAR—Return Disconnect Information

When an abortive release occurs, the transport user must immediately terminate data transfer and respond by invoking the TCLEAR service function. The purpose of the TCLEAR service is to return information associated with the disconnect. TCLEAR returns a disconnect reason code. The disconnect reason code is specific to the underlying transport protocol and should not be interpreted by application programs that intend to be independent of protocol.

```

*****
*  TRECVC COMPLETED WITH ERROR -- CHECK FOR DISCONNECT
*****
      USING TPL,1
TRECVERR CH 15,=AL2(TRFAILED)  ROUTINE FAILURE?
          BNE FATAL             IF NOT, NO RECOVERY
          CH 0,=AL2(TAINTEG)    DATA INTEGRITY ERROR?
          BNE NOTDISC          IF NOT, CAN'T BE DISCONNECT
          CLI TPLERRCD,TEDISCON DISCONNECT INDICATION?
          BNE NOTDISC          IF NOT, DON'T ISSUE TCLEAR
          TCLEAR EP=(9)        ACKNOWLEDGE DISCONNECT
          LTR 15,15            TCLEAR FAILED?
          BNZ FATAL            HOW CAN THAT BE?
          .
          . [connection is released, no data transfer is allowed]
          .
          TDSECT TPL          GENERATE TPL DSECT

```

Using TDISCONN and TCLEAR During Connection Establishment

Although TDISCONN and TCLEAR are generally invoked at the end of the data transfer phase, they can also be used during connection establishment. It was already mentioned that TREJECT is a special case of TDISCONN, issued by the server to reject a connection request. TDISCONN can also be issued by the client to revoke a connection request. In this case, a disconnect indication is presented to the server when it attempts to accept or reject a connection. The server must then invoke TCLEAR to receive the disconnect information. A sequence number is returned to identify the connection request that was revoked. A disconnect indication may also be issued if connection establishment is abandoned by the transport provider.

TRELEASE—Orderly Release Procedure

The orderly release procedure requires two steps by each transport user. The first user to complete data transfer initiates orderly release by invoking the TRELEASE service function. This function informs the transport provider (and peer transport user) that no more data is sent by the issuing transport user. The transport user that initiated the release must continue receiving data, and the peer transport user may continue sending data until all such data is transferred. At that time, the peer transport user invokes its equivalent of the TRELEASE function, indicating that it is now ready to release the connection. The connection is released only after both users have requested an orderly release and received the corresponding indication from the other user.

```
*****
* NO MORE DATA TO SEND--RELEASE CONNECTION GRACEFULLY
*****
RELEASE  TRELEASE EP=(9)          INITIATE CONNECTION RELEASE
          LTR 15,15                RELEASE STARTED?
          BNZ TRLSEERR             IF NOT, CHECK FOR DISCONNECT
          .
          . [receive data until release indication arrives]
          .
          TRELACK EP=(9)           ACKNOWLEDGE RELEASE INDICATION
          LTR 15,15                CONNECTION RELEASED?
          BNZ TRLSEERR             ERROR OCCURRED
```

TRELACK—Checking for Orderly Release

The release indication is presented to the transport user in a manner similar to all other API indications:

- An error is generated on the completion of the first TRECVC function after all data is received
- An asynchronous exit routine is scheduled if enabled by the transport user

In either case, the transport user must acknowledge the indication by invoking the TRELACK service function. TRELACK informs the transport provider that it has received all of the data and is aware of the pending release condition.

```
*****
* TRECVC COMPLETED WITH ERROR -- CHECK FOR ORDERLY RELEASE
*****
          USING TPL,1
TRECVCERR CH 15,=AL2(TRFAILED)    ROUTINE FAILURE?
          BNE FATAL                IF NOT, NO RECOVERY
          CH 0,=AL2(TAINTG)        DATA INTEGRITY ERROR?
          BNE NOTRLSE              IF NOT, CAN'T BE RELEASE
          CLI TPLERRCD,TERELEASE    ORDERLY RELEASE INDICATION?
          BNE NOTRLSE              IF NOT, DON'T ISSUE TRELACK
          TRELACK EP=(9)           ACKNOWLEDGE RELEASE INDICATION
          LTR 15,15                TRELACK FAILED?
          BNZ CHKDISC              PERHAPS DISCONNECTED
          .
          . [continue sending data until no more to send]
          .
```

```
TRELEASE EP=(9)          NOW RELEASE CONNECTION
LTR 15,15              CONNECTION RELEASED?
BNZ CHKDISC           IF NOT, CHECK FOR DISCONN.
. [graceful release is now complete]
.
TDSECT TPL             GENERATE TPL DSECT
```

The transport connection is not released until both functions are invoked by each transport user. The order in which they are invoked depends on which transport user initiates the release. Both users can initiate the release at the same time, and all race conditions are resolved by the API and the transport provider.

TRELACK can also be issued before the indication is received. In this case, the request remains pending and does not complete until a release indication is received from the transport provider. If the transport user is operating in synchronous mode, the task is suspended until the release indication arrives. Otherwise, the transport user is free to issue additional requests.

Note: This mode is useful when data transfer is uni-directional, and the transport user does not expect to receive any data.

Once TRELEASE is invoked, TSEND requests cannot be issued. Similarly, no TRECVC request can be issued once TRELACK is issued. There are no release data and reason codes associated with either function. The only parameters interpreted by TRELEASE and TRELACK are the common parameters, primarily the endpoint ID.

An orderly release can be interrupted (that is, aborted) by a disconnect indication. When this happens, the transport user must respond by invoking TCLEAR as described in [TCLEAR – Return Disconnect Information](#).

Connectionless-Mode Service

Connectionless-mode service does not require a connection for the transfer of data between two transport users. This mode of service is well suited to those applications where the interaction between two transport users is short-term, perhaps to transfer just a single unit of data, and where the overhead required to establish and release a connection may be prohibitive. Transaction-based processes characterized by simple request/response interactions are examples of applications where connectionless-mode service might be more appropriate.

Since a connection does not exist to identify the destination of data transfer and the options that affect the transfer, this information must be supplied with each service access. The source and destination are identified by their transport service access point address, just as they are for connection-mode service. However, each data unit transferred can have a different destination and may have no relationship at all to previous and subsequent data units. The underlying protocols that provide the connectionless service make no guarantee with respect to the order or duplication of data units. Nevertheless, they do guarantee that any data unit delivered is delivered intact and without corruption.

Connectionless-mode service has two phases:

- Local endpoint management (identical to connection-mode service in all but a few respects)
- Data transfer

Local Endpoint Management

The API service functions listed in [Connection-Mode Service](#) also apply to connectionless-mode. An endpoint must be opened that selects the communications domain and transport provider, and a local protocol address must be bound to the endpoint. At this time, the endpoint is ready for data transfer. After data transfer is complete, the endpoint can be unbound from its protocol address and closed. The miscellaneous functions that provide protocol information, manipulate options, and specify a user ID can also be used. In general, the local management of a connectionless-mode endpoint is identical to a connection-mode endpoint except for a few minor differences.

TOPEN—Opening an Endpoint

TOPEN is used to open an endpoint. Connectionless-mode service is selected by specifying CLTS as the service type, and the communications domain can be any of the domains described for connection-mode service. Thus, the TYPE and DOMAIN parameter select the transport provider and the protocol that provides the service. Alternatively, a protocol number or service ID can be used to make this selection. The APCB parameter identifies an opened APCB that, in turn, identifies the instance of the API to be used as the interface.

```
*****
* INITIALIZE CONNECTIONLESS-MODE ENDPOINT USING UDP PROTOCOL
*****
EPINI  TOPEN DOMAIN=INET,TYPE=CLTS,APCB=TUAPCB OPEN ENDPOINT
      LTR   15,15                ENDPOINT CREATED?
      BNZ   TOPENERR             IF NOT, GO TO ERROR ROUTINE
      USING TPL,1
      L     9,TPLEPID            LOAD NEW ENDPOINT ID
      DROP  1
      TBIND EP=(9),ADBUF=CLIENTPA,ADLEN=L'CLIENTPA,QLSTN=0,      +
            OPTCD=ASSIGN      ASSIGN TRANSPORT ADDRESS
      LTR   15,15                BIND SUCCESSFUL?
      BNZ   TBINDERR             IF NOT, GO TO ERROR ROUTINE
      .
      . [client can now send and receive datagrams]
      .
TUAPCB  APCB  AM=TLI,APPLID=EXAMPLE DEFINE TRANSPORT USER
CLIENTPA DS XL(LTPAINET)           CLIENT PROTOCOL ADDRESS
          TDSECT TPL,TPA,DOMAIN=INET GENERATE TPL AND TPA DSECTS
```

Any exit routines required to service the endpoint must be specified with the TOPEN function, if not specified when the APCB was opened.

These asynchronous protocol exits are supported for CLTS endpoints:

DATA Datagram received.
 DGERR Datagram error received.

The identifier returned by TOPEN must be used in all subsequent references to the endpoint. When the endpoint is no longer required, or control must be passed to another task or address space, the TCLOSE service function should be invoked.

The transport interface defines an inherent client/server relationship between two transport users when establishing a connection with connection-mode service. However, a similar relationship is not reflected in the definition or use of connectionless-mode service functions.

Note: It is the context of the application, not the transport interface that defines one transport user as a server and another as a client. There are similarities with connection-mode service (for example, the server is often passive and waits for a request from a client whose address is unknown in advance; the client is the active participant that initiates the interaction and contacts the server at a well-known address).

Using TBIND with Connectionless Mode

Binding of protocol addresses via the TBIND function closely parallels connection-mode service. The transport user in the role of a client generally requests the API to assign the local transport address, while the server specifies a well-known address of its choosing. However, connectionless endpoints must always be disabled, and the QLSTN parameter must be zero when issuing a TBIND request.

Terminating a Connectionless Mode Endpoint

Any protocol address bound to an endpoint can be unbound by invoking the TUNBIND function. Finally, the endpoint is terminated with the TCLOSE function.

```
*****
*   TERMINATE USE OF CONNECTIONLESS-MODE ENDPOINT
*****
EPTERM  TUNBIND EP=(9)           UNBIND PROTOCOL ADDRESS
        LTR    15,15             UNBIND SUCCESSFUL?
        BNZ    TUBNDERR          IF NOT, GO TO ERROR ROUTINE
        TCLOSE EP=(9)           CLOSE AND DELETE ENDPOINT
        .
        . [endpoint must not be referenced after closing]
        .
```

Using TADDR to Retrieve Addresses

The TADDR service can be used to retrieve local and remote protocol addresses.

Note: However, if not carefully used, the results may be incorrect.

When the LOCAL option code is specified to retrieve the local protocol address, the transport address returned is always the address currently bound to the endpoint, and the network address is the local network address through which the most recent datagram was transferred. When REMOTE is specified, the protocol address contains the transport and network address of the source of the last received datagram or the destination of the last sent datagram. If any send or receive requests are in progress at the time TADDR is executed, the results are unpredictable. The transport user should either quiesce data transfer before issuing a TADDR request, or rely only on addresses returned with other service functions.

Specifying Protocol Options

Protocol options can be negotiated with the TOPTION service. Any options negotiated remain in effect until changed with a subsequent TOPTION request. As an alternative, the transport user can specify protocol options with each datagram transmitted. Protocol options are protocol-dependent and should be avoided by application programs that may need to operate with other transport protocols at some future date. As usual, the TINFO service can be used to determine the characteristics and limits of the transport provider and the underlying protocol.

Data Transfer

A connectionless-mode endpoint bound to a local protocol address is immediately ready to send and receive data. Data is transferred as individual units of data, sometimes referred to as datagrams.

CLTS Data Transfer Functions

The following table lists the service functions supporting connectionless data transfer.

Function	Parameters	M/O	Description
TRECVERR	Endpoint ID Protocol Address Datagram Error Code	M OR MR	Receive pending datagram error indication.
TRECVFR	Endpoint ID Protocol Address User Data Residual Count	M OR MR MR	Receive a datagram and its accompanying source address
TSENDTO	Endpoint ID Protocol Address User Data Residual Count	M M M MR	Send datagram to designated destination.

SENDTO—Sending Outgoing Datagrams

Each outgoing datagram is sent by invoking the TSENDTO service function and must be accompanied by the protocol address of the destination transport user. The sending transport user may also specify protocol options (for example, quality of service parameters) that should be associated with the transfer of data. When the datagram arrives at the destination, the unit of data and its associated protocol options are delivered to the transport user. The size of the datagram is preserved. That is, a datagram is not split or combined with other datagrams.

Each data unit is treated individually in accordance with the quality of service parameters provided with the protocol options parameter. However, in the absence of protocol options, all data is treated the same. In particular, the API does not support normal and expedited data classes as it does for connection-mode, and the NORMAL and EXPEDITED option codes should not be indicated. Also, MORE and NOMORE have no meaning in the context of sending datagrams and, if indicated, are ignored.

```

* BOUNCE A DATAGRAM OFF OF UDP ECHO SERVER AT 127.0.0.1
*****
SENDECHO  TSENDTO EP=(9),ADBUF=SERVERPA,ADLEN=LTPAINET,DABUF=ECHOSEND,  +
          DALEN=ECHOLEN          SEND DATAGRAM TO ECHO PORT
          LTR 15,15              DATAGRAM SENT?
          BNZ TSNDTOER          IF NOT, GO TO ERROR ROUTINE
          .
          . [should probably set timer in case datagram is lost]
          .
RECVECHO  TRECVR EP=(9),ADBUF=SOURCEPA,ADLEN=LTPAINET,DABUF=ECHORECV,  +
          DALEN=ECHOLEN          RECEIVE ECHOED DATAGRAM
          LTR 15,15              DATAGRAM RECEIVED?
          BNZ TRCVFRER          IF NOT, GO TO ERROR ROUTINE
          CLC SOURCEPA,SERVERPA  FROM ECHO SERVER?
          BNE RECVECHO          IF NOT, GO RECV. SOME MORE
          USING TPL,1
          L 2,TPLDALEN          LOAD LENGTH OF RECEIVE DATA
          C 2,=A(ECHOLEN)      SAME AS AMOUNT SENT?
          BNE ECHOERR          IF NOT, LOG ERROR
          CLC ECHORECV(ECHOLEN),ECHOSEND DOES DATA MATCH?
          BNE ECHOERR          IF NOT, GO LOG IT
          B  SENDECHO          ELSE, SEND ANOTHER
          .
          .
SERVERPA  DC AL2(TDINET),AL2(7),AL1(127,0,0,1) SERVER PROTOCOL ADDR
SOURCEPA  DS XL(LTPAINET)      SOURCE PROTOCOL ADDRESS
ECHOSEND  DC C'NOW IS THE TIME FOR ALL GOOD DATAGRAMS ' ECHO-GRAM
          DC C'TO GO BACK FROM WHENCE THEY CAME'
ECHOLEN   EQU *-ECHOSEND
ECHORECV  DS XL(ECHOLEN+1)     RECEIVE BUFFER
          TDSECT TPL,TPA,DOMAIN=INET GENERATE TPL AND TPA DSECTS
* NOTE: LTPAINET and TDINET are defined in the
* TPA and TPL macro expansions, respectively

```

The datagram must be furnished to the API in one service access. That is, the datagram must be contained in a single transport interface data unit, and the EOM and NOTEOM option codes are not interpreted by the TSEND function. However, the data can be provided in direct or indirect format. The length of an outgoing datagram must be within the TIDU and TSDU send limits as defined in the TIB, obtained with the TINFO service.

TRECVR—Receiving Incoming Datagrams

The TRECVR service function receives incoming datagrams. A storage area must be provided for the datagram, and if the source address and accompanying protocol options are to be returned with the data, their respective storage areas must be identified.

When the TRECVR function completes:

- The data and source protocol address are returned in the storage areas provided
- Their lengths are updated to reflect the actual amount of data returned

A single datagram can be received with multiple service accesses. The EOM and NOTEOM option codes are asserted as appropriate to indicate when the end of the datagram is received. MORE is asserted to indicate that additional datagrams are available to be read. When MORE is asserted, the transport user can issue a TRECVR function and know that it completes immediately without suspending the issuing task for an extended period.

The API allocates internal send and receive buffers for moving data between address spaces, and manages these buffers in a manner similar to connection-mode service. A residual byte count is returned when either a TSENDTO or TRECVR function completes. Multiple instances of these functions may be invoked without waiting for the first to complete. The limits defined at installation time can be retrieved with the TINFO service, and the values in effect for the endpoint can be changed using the TOPTION service.

A TSENDTO request can complete as soon as the datagram is passed to the transport provider. Therefore, subsequent protocol errors such as nonreachable destinations must be signaled to the transport user later. If the transport user enabled the datagram error exit, its exit routine is scheduled when a datagram error indication arrives. Otherwise, the next TSENDTO or TRECVR request completes with an error. In either case, the transport user must receive the indication by invoking the TRECVER service function. The destination address of the datagram, along with any protocol options specified, is returned to the transport user. A protocol-dependent error code is also returned.

Note: Application programs that intend to remain protocol-independent should not interpret this error code and should use it for diagnostic purposes only.

```
*****
*  TRECVR COMPLETED WITH ERROR --CHECK FOR DATAGRAM ERROR
*****
      USING TPL,1
TRCVFRER  CH  15,=AL2(TRFAILED)      ROUTINE FAILURE?
          BNE  FATAL                  IF NOT, NO RECOVERY
          CH  0,=AL2(TAINTG)         DATA INTEGRITY ERROR?
          BNE  NOTDGERR IF NOT,      CAN'T BE DATAGRAM ERROR
          CLI  TPLERRCD,TEPROTO      DATAGRAM ERROR INDICATION?
          BNE  NOTDGERR              IF NOT, DON'T ISSUE TRECVR
          TRECVR EP=(9),ADBUF=DGERRPA,ADLEN=L'DGERRPA
*
          LTR  15,15                  GET ERROR INFO
          BNZ  FATAL                  TRECVR FAILED?
          L   2,TPLDGERR             WHOOPS, HOW CAN THAT BE?
                                     LOAD DATAGRAM ERROR CODE
          .
          . [the dest. addr and error code should be logged]
          .
DGERRPA   DS   XL(LTPAINET)          DATAGRAM ERROR PROTOCOL ADDRESS
          TDSECT TPL,TPA,DOMAIN=INET GENERATE TPL DSECT
```

Connectionless Service with Associations

The interaction between two connectionless-mode transport users may be more involved than just a simple request/response transaction, and a client can be engaged in a conversation with a server for an extended period. In this case, the transport user may prefer the benefits of a long-term connection afforded by connection-mode service, but prefer the efficiency and simplicity of connectionless-mode data transfer. The API accommodates this by providing a service that is a mixture of these two modes. The API uses the term *association* to distinguish this type of service from the true transport connection supported by connection-mode service and the pure form of connectionless-mode service.

An association is used to communicate with a connectionless-mode transport user for an extended period. The association is established by the local transport user in exactly the same manner as a real transport connection is established using connection-mode services. The distinction between client and server is also reflected in the manner in which the association is established. After an association exists, data transfer proceeds as if a connection was established. Therefore, after data transfer is complete, the association must be released as if it were a real connection. Simply stated, establishing an association with another transport user allows an extended exchange of data between a client and server without having to provide or process protocol addresses with every transfer request.

The API performs these services using the standard services of the connectionless mode transport provider (for example, the local transport user issues connection-mode service requests and the API simulates the services of a connection-mode provider). The real transport provider is used only for data transfer, and the existence of an association is transparent to the remote transport user. Since the underlying protocol is connectionless, the characteristics of data transfer are those of connectionless-mode service – the data stream between the associated transport users is a sequence of datagrams, some of which may be lost or duplicated.

ASSOC—Requesting Association-Mode Service

The association-mode service must be requested when the endpoint is opened. This is done by including the ASSOC sublist parameter when CLTS is requested as the service type. All other parameters of the TOPEN request apply as if a connection-mode endpoint is being created. In particular, the exits supported for associations are the same as those supported for connections. Once the endpoint is opened, local endpoint management proceeds as if a transport connection is going to be established, except that transport addresses must be consistent with the underlying protocol.

That is, if the endpoint was created in the internet (INET) domain:

- The underlying protocol is UDP
- The transport addresses must be UDP port numbers and not TCP port numbers

```
*****
*   CREATE AN ENDPOINT FOR SERVER-MODE ASSOCIATIONS
*****
      USING TPL,1
EPINIT  TOPEN  DOMAIN=INET,TYPE=(CLTS,ASSOC),APCB=TUAPCB
*
      LTR    15,15          OPEN ENDPOINT
      BNZ    TOPENERR      ENDPOINT CREATED?
      L      9,TPLEPID     IF NOT, GO TO ERROR ROUTINE
      TBIND  EP=(9),ADBUF=SERVERPA,ADLEN=LTPAINET,QLSTN=1,
      OPTCD=USE           LOAD NEW ENDPOINT ID
      LTR    15,15          BIND SERVER TRANSPORT ADDRESS
      BNZ    TBINDERR      BIND SUCCESSFUL?
TUAPCB  APCB  AM=TLI,APPLID=EXAMPLE  DEFINE TRANSPORT USER
```

Establishing Client Associations

Establishing an association to a server-mode transport user is straightforward. The TCONNECT service function is invoked giving the (connectionless) protocol address of the server and any protocol options to associate with data transfer. This information is retained by the API and used with subsequent service primitives issued to the transport provider.

```
*****
* ESTABLISH A CLTS ASSOCIATION WITH TFTP SERVER AT 127.0.0.1
*****
EPINIT   TOPEN DOMAIN=INET,TYPE=(CLTS,ASSOC),APCB=TUAPCB
*
        LTR 15,15          OPEN ENDPOINT
        BNZ TOPENERR      ENDPOINT CREATED?
        USING TPL,1       IF NOT, GO TO ERROR ROUTINE
        L 9,TPLEPID       LOAD NEW ENDPOINT ID
        DROP 1
        TBIND EP=(9),ADBUF=CLIENTPA,ADLEN=L'CLIENTPA,QLSTN=0,      +
        OPTCD=ASSIGN      ASSIGN TRANSPORT ADDRESS
        LTR 15,15          BIND SUCCESSFUL?
        BNZ TBINDERR      IF NOT, GO TO ERROR ROUTINE
        TCONNECT EP=(9),ADBUF=SERVERPA,ADLEN=LTPAINET MAKE ASSOCIATION
        LTR 15,15          ASSOCIATION INITIATED?
        BNZ TCONNERR      IF NOT, GO TO ERROR ROUTINE
        TCONFIRM EP=(9)   WAIT FOR CONFIRMATION
        LTR 15,15          ASSOCIATION CONFIRMED?
        BNZ TCONFERR      IF NOT, GO TO ERROR ROUTINE
        .
        . [the endpoint is now ready for data transfer]
        .
TUAPCB   APCB AM=TLI,APPLID=EXAMPLE DEFINE TRANSPORT USER
SERVERPA DC AL2(TDINET),AL2(69),AL1(127,0,0,1) TFTP SERVER ADDR
CLIENTPA DS XL(LTPAINET) CLIENT PROTOCOL ADDRESS
        TDSECT TPL,TPA,DOMAIN=INET GENERATE TPL AND TPA DSECTS
* NOTE: LTPAINET and TDINET are defined in the
* TPA and TPL macro expansions, respectively
```

Example

A subsequent TSEND request causes the data to be sent as a datagram to the remote transport user using the protocol address supplied to the TCONNECT function. Incoming datagrams are filtered using the local protocol address, and only those that match the filter are delivered to the transport user; everything else is discarded. The transport provider confirms the association by issuing a confirm indication, which must be received by invoking the TCONFIRM service. The transport user's confirm indication exit routine is scheduled, if appropriate.

Establishing Server Associations

A transport user operating in server mode can be single-threaded or multithreaded. The server listens for connect indications by enabling the appropriate exit routine or by invoking the TLISTEN service function.

A connect indication is generated whenever a datagram is received from a source for which no association exists. The datagram is queued until the connect indication is accepted or rejected by a TACCEPT or TREJECT request. The value of QLSTN, specified in the TBIND request that enabled the endpoint, defines the number of datagrams from unique sources that can be queued at one time. The association can be accepted to a new endpoint if the server is operating in multithreaded mode. Once the association is established, incoming datagrams are routed to the appropriate endpoint and are received by invoking the TRECVC function.

```
*****
* ESTABLISH A CLTS ASSOCIATION AS A SINGLE-THREADED SERVER
*****
      USING TPL,1
LISTEN  TLISTEN EP=(9),ADBUF=CLIENTPA,ADLEN=L'CLIENTPA INITIATE LISTEN
      LTR 15,15          DATAGRAM RECEIVED?
      BNZ TLSTNERR      IF NOT, GO TO ERROR ROUTINE
      L 7,TPLSEQNO      LOAD SEQUENCE NUMBER
      TACCEPT EP=(9),SEQNO=(7) ACCEPT CONNECT INDICATION
      LTR 15,15          ASSOCIATION ESTABLISHED?
      BNZ TACPTERR      IF NOT, GO TO ERROR ROUTINE
      .
      . [the endpoint is now ready for data transfer]
      .
ENDDATA TDISCONN EP=(9)          SIMULATE DISCONNECT
      LTR 15,15          ASSOCIATION TERMINATED?
      BZ LISTEN          IF SO, LISTEN FOR NEXT CLIENT
      .
      . [handle disconnect error]
      .
SERVERPA DC AL2(TDINET),AL2(69),AL4(0) TFTP TRANSPORT ADDRESS
CLIENTPA DS XL(LTPAINET)          CLIENT PROTOCOL ADDRESS
      TDSECT TPL,TPA,DOMAIN=INET GENERATE TPL AND TPA DSECTS
```

Data Transfer with Associations

All data transfer is executed with TSEND and TRECVC functions.

The expedited data option is not supported. However, since datagram boundaries are preserved, the notion of TSDU boundaries is supported, and EOM/NOTEOM is set accordingly for inbound data. MORE indicates the presence of another datagram on input and is ignored on output.

Releasing Associations

The association is released in the same fashion as a connection. Since all connection-oriented functions are simulated by the API, orderly release is always available and need not (and should not) be requested when the endpoint is opened.

Invoking TRELEASE causes a release indication to be generated after the last available datagram is received. The indication must be acknowledged with TRELACK. Since an association has no end-to-end significance, the remote transport user cannot initiate the release of an association. Therefore, the server must have some method for determining when the client no longer requires service and then terminate the association.

Local Endpoint Control

This section describes local endpoint control processing and the functions it uses.

The following table lists local API functions used to control processing at an endpoint

Function	Parameters	M/O	Description
TCHECK	Endpoint ID	M	Waits for completion and schedule error recovery routine if completed abnormally.
TERROR	Endpoint ID	M	Analyzes error and generates messages in WTO list format.
TEXEC	Endpoint ID	M	Reexecutes previous function.
TSTATE	Endpoint ID	M	Gets current endpoint state.

Function Differences

These functions generally differ from the service functions described in the previous sections in these major respects:

- Control functions are processed within the transport user's local system
- Control functions have the address of a parameter list (TPL) used with a previous service request as their only operand

The important distinction for these control functions is that they are not a new request for service to be processed by the API or the transport provider. Rather, they represent a particular control process to be performed in connection with a previous service request.

Such control processes include:

- Synchronizing with the completion of an outstanding request
- Scheduling synchronous error handling routines
- Generating diagnostic error messages
- Reexecuting a previous request
- Determining the state of an endpoint after the completion of a service function

TCHECK

Note: TCHECK is perhaps the most widely used and most important control function.

The sole parameter to the TCHECK function is the address of an active TPL associated with a previous service request.

TCHECK performs these important functions:

- Synchronizes the application program with the completion of the service request
- Sets the TPL inactive (and reusable for another request)
- Schedules the appropriate error recovery routine if the request completes abnormally

When application programs are operating in synchronous mode, TCHECK is executed automatically at the end of each service request. In the previous programming examples, the default operating mode was synchronous. Asynchronous mode must be requested by asserting the ASYNC option code when a service request is issued. In this case, control is returned immediately to the application program after the request has been accepted, and the application program can perform other processing. However, at some point the application program must execute a TCHECK function using the TPL address of the pending request. Thus, in asynchronous mode, TCHECK must be invoked for every instance of an API service request. TCHECK is discussed in more detail in the chapter "Program Synchronization and Control."

```

*****
* LISTEN FOR CONNECTION REQUESTS IN ASYNCHRONOUS MODE
*****
LISTEN  TLISTEN EP=(9),ADBUF=CLIENTPA,ADLEN=L'CLIENTPA,ECB=INTERNAL,  +
        OPTCD=ASYNCLISTEN FOR CONNECTION REQUESTS
        LTR 15,15      REQUEST ACCEPTED?
        BNZ TLSTNERR   IF NOT, GO TO ERROR ROUTINE
        ST  1,LISTENPL ELSE, SAVE TPL ADDRESS
        .
        . [application program can perform other processing]
        .
        L  1,LISTENPL   GET LISTEN TPL ADDRESS
        TCHECK TPL=(1)  WAIT FOR CONNECT INDICATION
        USING TPL,1
        L  7,TPLSEQNO   LOAD SEQUENCE NUMBER
        DROP 1
        .
        . [SYNAD routine handles TLISTEN completion errors]
        .
LISTENPL DS  F          LISTEN TPL ADDRESS
CLIENTPA DS  XL(LTPAINET) ADDRESS OF CLIENT
          TDSECT TPL,TPA,DOMAIN=INET GENERATE TPL AND TPA DSECTS

```

TERROR—Abnormally Completed Service Requests

When a request for service completes abnormally, information returned in the TPL defines the particular error. There are numerous reasons a request may complete abnormally, ranging from programming logic errors to network and protocol malfunctions. To assist the user in processing such errors, the API provides the TERROR function, which analyzes the error and generates an informative message.

The parameter supplied to the TERROR function must be the address of an inactive TPL that completed abnormally. The information contained in the TPL is analyzed by TERROR, and an error message is generated in WTO list format. This error message can be displayed to an interactive user or system operator or logged for later diagnosis by a programmer.

```

*****
* SYNCHRONOUS ERROR RECOVERY ROUTINE
*****
        USING SYNADX,12
SYNADX  LR 12,15      ESTABLISH BASE ADDRESS
        LR  2,0       SAVE RECOVERY ACTION CODE
        LR  3,1       SAVE TPL ADDRESS
        TERROR TPL=(1) GENERATE DIAG. ERROR MSG.
        LTR 15,15     MESSAGE RETURNED?
        BNZ SKIPWTO   IF NOT, SKIP WTO
        LR  4,0       SAVE WTO LIST ADDRESS
        USING TEM,4
        WTO MF=(E,TEMWTO) WRITE MESSAGE TO OPERATOR
        L  0,TEMSL    LOAD LENGTH AND SUBPOOL
        FREEMAIN RU,LV=(0),A=(4) RELEASE MSG. STORAGE AREA
        DROP 4
SKIPWTO DS  OH
        .
        . [perform recovery processing]
        .
        TDSECT TEM          GENERATE TEM DSECT

```

TEXEC—Executing a Fully-initialized TPL

TEXEC executes a TPL that is fully initialized, including the function code associated with a given service request. TEXEC is typically used in an error recovery routine to reexecute a previous request, or when the application program wants to bypass the API macro instructions and manipulate the TPL directly.

Note: Strictly speaking, TEXEC is not a local function, since it invokes an API service, but it is presented here since it does not fit in any of the function groups previously discussed. In fact, TEXEC can be used to execute any of the previously discussed service functions and, therefore, belongs in all groups.

TSTATE—Return Endpoint State

TSTATE is the last control function. Its primary use is to get state information associated with a given endpoint. The endpoint is identified by providing the address of any TPL used with any service request. Since the TSTATE function also checks to see if a TPL is inactive, and if not, whether the request is complete, TSTATE also can be used to poll the status of a pending service function.

Declarative Macro Instructions

Declarative macro instructions are macro instructions that never generate any executable code. They are generally used to define data areas used by other macro instructions.

These are the declarative macro instructions supported by the API.

APCB	Generates Application Program Control Block (APCB) used by AOPEN and ACLOSE session services.
TDSECT	Generates dummy control sections (DSECTs) for API user-level control blocks.
TEVNTLST	Generates list of protocol event ECBs or exits. Referenced by TOPEN request.
TEXTLST	Generates exit list referenced by AOPEN and TOPEN functions.
TPL	Generates a Transport Service Parameter List (TPL).

The APCB Macro Instruction

The APCB macro instruction generates the APCB referenced by the AOPEN, ACLOSE, and TOPEN service functions. Any fields of the APCB that normally may be set by the application program can be specified with the APCB macro instruction. The list form of the macro instruction generates an APCB in-line with the macro instruction. The APCB is modified by the AOPEN service function and must not be in protected storage. There is no reentrant (that is, remote list) form of the APCB macro instruction, and if the application program is reentrant, the APCB must be moved into dynamic storage before it is opened. An alternate form of the APCB macro instruction generates a DSECT that maps the fields of the APCB for direct program manipulation.

The TDSECT Macro Instruction

The TDSECT macro instruction can be used to generate a DSECT for all other user-provided data structures interpreted by the API.

The following is a list of the defined data structures.

TEM	Transport Service Error Message.
TIB	Transport Service Information Block.
TPA	Transport Protocol Address.
TPL	Transport Service Parameter List.
TPO	Transport Protocol Options.
TSW	Transport Endpoint State Word.
TUB	Transport Endpoint User Block.
TXL	Transport Endpoint Exit List.
TXP	Transport Endpoint Exit Parameters.

The TEXTST Macro Instruction

The TEXTST macro instruction is used to define an exit list referenced by the AOPEN or TOPEN service functions. A positional parameter defines which service function the exit list is used with, it also defines the maximum length of the exit list and its contents. Only a subset of the exits supported in an AOPEN exit list can be specified in a TOPEN exit list.

Each keyword operand of the TEXTST macro instruction defines the entry point of an exit routine that may be scheduled by the API to process certain asynchronous events. See the *TCPaccess Assembler API Macro Reference* for more information on the TEXTST macro instruction.

Here is an example of this macro instruction:

```
*****
* DEFINE APPLICATION PROGRAM CONTROL BLOCK REFERENCING TEXTST
*****
TUAPCB  APCB  AM=TLI,APPLID=EXAMPLE,EXLST=TUEXLST  GENERATE APCB
TUEXLST  TEXTST  AOPEN,SYNAD=SYNADX,LERAD=LERADX  SPECIFY EXIT ROUTINES
```

The TEVENTLST Macro Instruction

The TEVENTLST macro instruction is used to define ECBs to post for protocol event notification. TEVENTLST supports protocol event exits as well. TEVENTLST follows the same rule as TEXTST, except there is no TOPEN, AOPEN, APEND, SYNAD or LERAD parameter.

Note: Unlike TEXTST, which may be referenced on TOPEN or AOPEN, only TOPEN may reference TEVENTLST.

With TEVENTLST, each event keyword parameter takes two subparameter values:

- The first subparameter value is the address of the exit routine or ECB associated with the event
- A second subparameter is optional and is used to designate whether the first subparameter value is the address of an exit routine (EXIT) or the address of an ECB (ECB).

If the second subparameter value is omitted, the default value is EXIT.

Here is an example of this macro instruction:

```
*****
*  DEFINE  TEVNTLST
*****
      TOPEN  DOMAIN=INET ,TYPE=COTS ,APCB=TUAPCB ,          +
            EVENTLST=TUEVLST
      .
      .
TUAPCB  APCB  AM=TLI ,APPLID=EXAMPLE
TUEVLST TEVNTLST CONNECT=(CONN ,EXIT) , CONNECT INDICATION EXIT  +
            DATA=(DATAECB ,ECB) ,   DATA INDICATION ECB      +
            RELEASE=RELX           RELEASE INDICATION EXIT
```

The TPL Macro Instruction

The TPL declarative macro instruction can be used to generate the TPL for any service function other than TOPEN. The TPL macro instruction supports both the in-line and remote list forms. The TPL can also be generated by the list forms of the other API service functions.

Example

The TPL for a TBIND request can be generated with either the TBIND or TPL macro instructions. If generated with the TBIND macro instruction, the function code in the TPL is initialized for the TBIND service. Otherwise, the function code must be supplied later.

Another difference between the TPL macro instruction and function-specific macros is that TPL supports all possible TPL parameters, whereas function-specific macros support only the parameters valid for the specific function.

Endpoint States and Function Sequences

An address space can contain several transport users, and any given transport user can open several endpoints. The maximum number of transport users and endpoints is ultimately limited by API resources. It can be further restricted by limits defined during installation or by limits imposed by a particular transport provider. The activity on one endpoint generally is unrelated to the activity on any other endpoint. However, for a given endpoint, API service functions must be executed in a prescribed sequence. This sequence parallels the phases of service previously described in this chapter.

Endpoint Functions

During the lifetime of an endpoint, the particular functions that can be executed at any given moment are determined by:

- The service mode of the endpoint
- Characteristics of the transport provider
- The current state of the endpoint

The API service functions that apply to each service mode have been discussed in detail. The characteristics of the transport provider that apply to selected functions have also been discussed, and whether or not a particular transport provider possesses these characteristics can be determined at runtime by examining the TIB. This chapter concludes with a discussion of endpoint states and how they affect the execution sequence of API service functions.

Endpoint States

The current state of an endpoint is maintained in a data structure allocated by the API. This data structure is located via the endpoint identifier returned by `TOPEN`, and as such, all requests for service that reference the endpoint must be accompanied by this identifier. If the wrong endpoint identifier is provided, or has been corrupted, the request fails.

As API functions are executed, the endpoint transitions from one well-defined state to another. In some cases, the next state is the current state, and in other cases, it is a new state. However, at any given moment, the endpoint is in one of nine states.

The following table defines these states and how they relate to the service modes for which they are valid.

State Value	State Name	Service Type	Description
0	TSCLOSED	COTS CLTS	Closed. The endpoint is closed (nonexistent) or in the process of opening.
1	TSOPENED	COTS CLTS	Opened. The endpoint is opened, but no local protocol address was bound to the endpoint.
2	TSDSABLD	COTS CLTS	Disabled. A local protocol address is bound to the endpoint, and the endpoint is disabled for queuing incoming connection requests (QLSTN=0). If the service mode is connectionless, the endpoint is ready for data transfer.
3	TSENA BL D	COTS (CLTS,ASSOC)	Enabled. A local protocol address was bound to the endpoint, and the endpoint is enabled for queuing incoming connection requests (QLSTN>0).
4	TSINCONN	COTS (CLTS,ASSOC)	Connect-indication-pending. One or more connect indications were received that have not been accepted or rejected.
5	TSOUCONN	COTS (CLTS,ASSOC)	Connect-in-progress. An outgoing connection is in progress, and the endpoint is waiting connect confirmation.
6	TSCONNCT	COTS (CLTS,ASSOC)	Connected. A connection (or association) was established and the endpoint is ready for data transfer.
7	TSINRLSE	(COTS,ORDREL) (CLTS,ASSOC)	Release-indication-pending. An orderly release was received and acknowledged, and the endpoint may continue sending data.
8	TSOURLSE	(COTS,ORDREL) (CLTS,ASSOC)	Release-in-progress. An orderly release was initiated, and the endpoint may continue receiving data until released by the remote transport user.

The value of the current endpoint state can be obtained by the TSTATE control function and is returned within a 32-bit state word containing other status information. The structure and contents of this word is mapped by the Transport Endpoint State Word (TSW) DSECT, which can be generated by the TDSECT macro instruction. The state names listed in this table correspond to the symbols defined by the TSW DSECT.

State Transitions

State transitions occur on the successful completion of an API service function. If a function completes abnormally, the current state is not changed. The appendix “Endpoint State Transitions” lists all possible state transitions caused by any API service function.

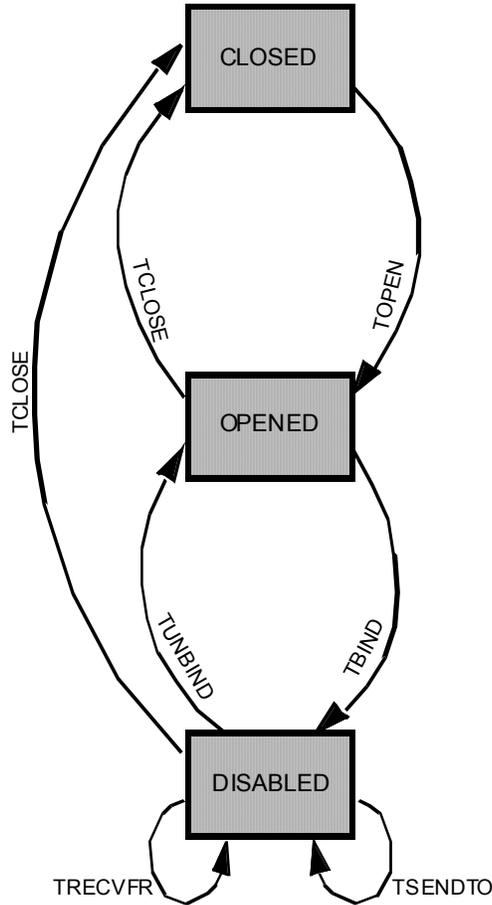
The following diagrams summarize the state transitions for connection-mode and connectionless-mode service.

Note: Due to the many variations possible, only the common transitions are shown. Use the appendix “Endpoint State Transitions” as the final authority for determining when a particular function can be invoked and what state transitions occur as the result of its successful execution.

- An endpoint can enter the enabled state from the disabled state; this state transition is not shown in the interest of keeping the diagram less cluttered. At this point, the client can initiate a connection to the server, and the server can receive connect indications from clients.
- A TCONNECT request successfully executed by the client causes the endpoint to enter the connect-in-progress state. The server receives the connect indication via a TLISTEN function, which transits the endpoint to the connect-indication-pending state.
- If the connect indication is accepted (TACCEPT), then the endpoint enters the connected state.
- If the connect indication is rejected (TREJECT), then the endpoint returns to the enabled state.
- Subsequent arrival of the connect confirmation causes a TCONFIRM request to complete, leaving the endpoint in the connected state. A disconnect indication received by TCLEAR, or a disconnect initiated by TDISCONN, causes the endpoint to return to the disabled state.
- An endpoint in the connected state is ready for data transfer and loses its distinction with respect to client or server modes of operation. TSEND and TRECVC requests may be executed indefinitely, each leaving the endpoint in its connected state.
- When data transfer is complete, either the client or server can initiate connection release. The transport user initiating release does so by executing the TRELEASE function, causing the endpoint to enter the release-in-progress state. While in this state, the endpoint can continue receiving data.
- When the peer transport user receives a release indication, it must acknowledge receipt by invoking the TRELACK service function. Successful completion of this request transits the endpoint to the release-indication-pending state, and the endpoint is able to continue sending data.
- When the complementary TRELEASE or TRELACK functions have been executed, completing release of the connection, the respective endpoints return to their original state prior to establishment of the connection.
- This state transition diagram also shows that a connection can be released abruptly by issuing a TDISCONN request or by receiving a disconnect indication via the TCLEAR function. In either case, the endpoint returns to the state it was in after the protocol address was bound. In addition, TCLOSE can be issued at any time, returning the endpoint to the closed state.

Connectionless-Mode States This diagram shows the state transitions for connectionless-mode service.

State Transitions for Connectionless-mode Services



This state diagram reflects the simplicity of connectionless-mode service, which uses only closed, opened, and disabled states.

All data transfer takes place in the disabled state. O

- On completion of data transfer, the endpoint can be closed – returning it to the closed state
- The local protocol address can be unbound – returning the endpoint to the opened state

Establishing an association with a CLTS endpoint has similar state transitions to connection-mode.

Function Sequences

Issuing a request when the endpoint is not in a proper state causes the request to complete with an error. In addition, issuing a request for service when a previous request, issued on the same endpoint, has not completed, generally causes an error. However, there are some important exceptions that let some functions overlap or be preempted by subsequent requests.

This is a brief list of these exceptions:

- Multiple TSEND, TSENDTO, TRECVC, and TRECVCFR requests can be issued subject to the limits defined in the TIB or negotiated with a TOPTION request
- One TRELEASE request can be issued while one or more TSENDS are pending, and one TRELACK request can be issued while one or more TRECVCs are pending
- TRETRACT can (and should be) issued while a TLISTEN request is pending
- One TDISCONN or TCLEAR request can be issued at any time except when a TCLOSE request is pending
- One TCLOSE request can be issued at any time

Regardless of the function requested, a TPL must not be reused until the previous request with which it is associated completes. If multiple requests are pending at any point in time, each must be associated with a different TPL. Techniques that can be used to synchronize with the completion of a request are presented in the chapter “Program Synchronization and Control.”

Under rare circumstances, the API may not be able to accept a request for service because of a temporary resource shortage, even if all function sequencing rules are obeyed. Should this happen, the request completes with an error indicating the resource shortage and should be reissued after some delay. However, it is much more likely that the transport provider will exhaust its resources before the API does.

Program Synchronization and Control

This chapter discusses how the application program synchronizes with the transport provider and controls the execution sequence of Unicenter TCPAccess API macro instructions. The information presented in this chapter together with the concepts and facilities presented in the preceding chapter provide the necessary background for understanding how the macro instructions operate. For detailed information about each API macro instruction, refer to the *Unicenter TCPAccess Communications Server Assembler API Macro Reference*.

This chapter discusses the following topics:

- [Task Synchronization Requirements](#) – Describes how to synchronize with the completion of a TPL-based service request and overlap endpoint processing with other application program activities
- [Modes of Operation](#) – Describes the various modes of operation for TPL-based service requests and provides detailed information on synchronization characteristics of the API macro instructions
- [Specifying and Using Exit Routines](#) – Describes how to specify, code, and use exit routines for processing various asynchronous events
- [Handling Errors and Special Conditions](#) – Describes how to detect and handle errors and other special conditions
- [Application Program Organization](#) – Describes facilities for program control and synchronization
- [Multitasking Operation Rules](#) – Describes the rules for using the API in a multitasking environment
- [Multiple Address Spaces](#) – Describes the facility provided to pass sessions from one address space to another
- [24-Bit and 31-Bit Addressing](#) – Describes use of the API with 24-bit and 31-bit addressing

Task Synchronization Requirements

The API must be capable of providing service within a variety of application program environments; single-threaded programs serving a single user must be supported as well as multithreaded programs supporting several users simultaneously.

In the latter case, such programs may be:

- Multi-task or multi-address-space
- Single task with a built-in scheduling algorithm for servicing individual user requests (for example, CICS)

In addition, the API must support the capability of overlapping network I/O with other application program processing (for example, disk I/O).

An important characteristic of the API design is, at the option of the application program, no transport service request causes the issuing task to be suspended. As a result, the processing of service requests can be easily overlapped with other application program activities, and when executing on a multi-processor, can actually be executed in parallel. To support this requirement, the API provides a mechanism for initiating a service request and synchronizing with its subsequent completion under total control of the application program.

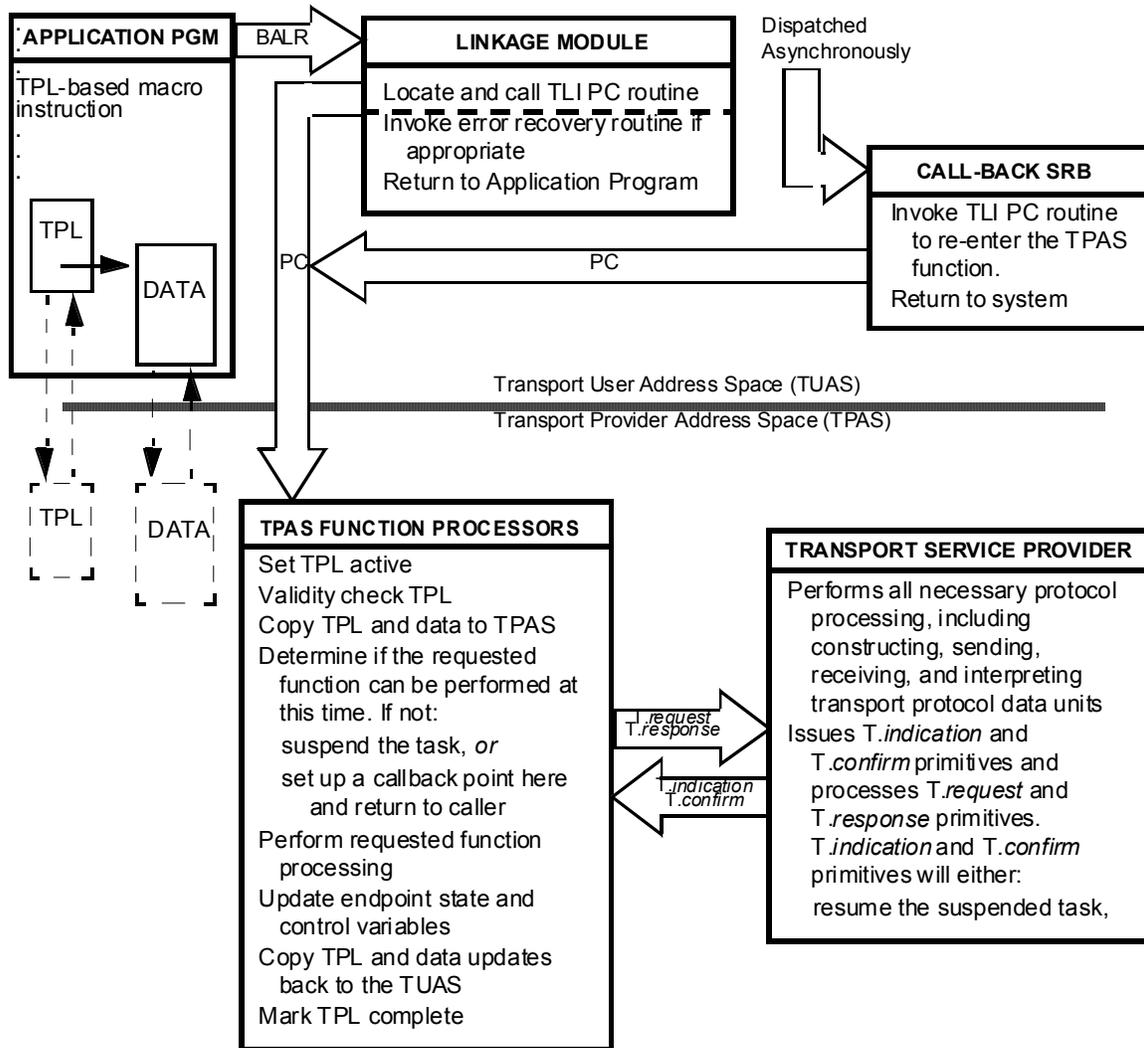
The synchronization mechanisms implemented by the API are similar to those of VTAM, and use standard MVS facilities. Thus, a programmer who is familiar with VTAM assembler language programming has already been introduced to most of the principles presented in this section. However, it is suggested that such programmers read this section as a review.

Typical Processing Flow

A service request requires processing in two MVS address spaces. The Transport User Address Space (TUAS) is the address space of the application program that initiates the request. The Transport Provider Address Space (TPAS) is the address space containing the API subsystem and the transport service provider that eventually processes the request. In some cases, the transport provider may reside in an external device (for example, a specialized network I/O processor), and the transport provider depicted in the following diagram is merely a stub that communicates with the external device. In either case, all other entities are present, and the actual location of the transport provider is transparent to the application program.

Execution Flow

The following diagram shows the typical processing flow for a transport service



Service Request
Initiation

All service requests are initiated by executing an API macro instruction.

This list describes the steps included in this process:

- The API macro expansion generates executable instructions that call a TUAS function interface that was loaded into common storage.
- The TUAS function interface locates the PC number for the TPAS interface and issues the PC instruction. The TPAS PC routine is a space-switched PC. The TPAS becomes the primary address space, although execution continues under control of the application program dispatchable unit.
- The TPAS PC routine locates and calls a TLI function-dependent processing routine (TPAS function processor).
- The TPAS function processor validates the request parameters and copies the TPL and associated data into the API address space.
- The TPAS function processor calls the respective transport provider routines (that is, connection management and data transfer routines).
- Depending upon the specific API request or the existing conditions of the endpoint at the time, the request can either complete immediately, or it may need to wait for a protocol event such as the arrival of data or a connect request.
- When a protocol event must occur before completing the request, either the application's dispatchable unit is suspended or the request is queued internally and control is returned to the application program. The requested mode of service (synchronous or asynchronous) determines the disposition of the event-pending request.

When the anticipated protocol event occurs, suspended synchronous requests are resumed. An SRB routine is scheduled into the application address space to redrive the internally queued request via the TPAS PC interface.

During the time that an asynchronous request is queued awaiting a protocol event, the application program may perform additional processing. Eventually, the application must issue a TCHECK to resynchronize with the request.

- The TPL and associated data areas in the application address space are updated.
- The application is notified of the completed request. For synchronous requests, this simply consists of returning control to the application. For asynchronous requests, completion processing depends if the request specified ECB or EXIT for completion notification.
- If ECB was specified, a cross-memory post is executed.

- If EXIT was specified, the exit routine address and parameter are placed into a queue element for processing by the TUAS IRB routine. The IRB routine must also be scheduled if it is not already active. Synchronization modes are discussed in [Modes of Operation](#).

Modes of Operation

The mode of operation affects how processing proceeds after a service request is accepted and scheduled for processing by the API address space. The mode of operation is selected individually for each request by indicating an option code when the macro instruction is executed.

This section describes the various modes of operation for TPL-based service requests and provides detailed information on synchronization characteristics of API macro instructions.

Example

OPTCD=SYNC indicates synchronous operation, and causes the request to complete before control is returned to the application program.

OPTCD=ASYNC indicates asynchronous operation, and causes control to be returned prior to completion if the request must await a protocol event.

Processing that precedes acceptance of a service request, and all processing within the API address space, is identical for both modes of operation. When a service request is issued, the TPL is set active and its contents are checked for validity. If the request is valid and the state of the endpoint is acceptable, the request is scheduled for processing by the API address space. When the request completes, the TPL is posted complete. The TPL remains active during this entire period, and if a subsequent request attempts to use the TPL, it completes immediately with an error.

Operating Mode Differences

The difference between operating modes arise after the request is issued and only affects processing in the application program’s address space.

- For synchronous mode, the TPAS function interface waits for the request to complete, and sets the TPL inactive before returning to the application program.
- For asynchronous mode, the TPL is left in its active state when control is returned. It is the responsibility of the application program to wait for completion and cause the TPL to be set inactive by executing a TCHECK macro instruction.

Execution of the TCHECK macro instruction is implicit in synchronous mode, and explicit in asynchronous mode.

Synchronization
Characteristics of
Macro Instructions

The following table lists each mode of operation. Operation modes are discussed in further detail in the [Synchronous Operation](#) examples.

Macro Instruction	Transport Provider Primitive Required For Completion	Notes
ACLOSE		1
AOPEN		
APCB		
TACCEPT	T-CONNECT.response	2
TADDR		
TBIND		
TCHECK		3
TCLEAR		
TCLOSE		4
TCONFIRM	T-CONNECT.confirm	
TCONNECT	T-CONNECT.request	
TDISCONN	T-DISCONNECT.request	
TDSECT		
TERROR		
TEXEC	Dependent on function code	
TEXTST		

Macro Instruction	Transport Provider Primitive Required For Completion	Notes
TINFO		
TLISTEN	T-CONNECT.indication	
TOPEN		
TOPTION		
TPL		
TRECV	T-DATA.indication or T-EXPEDITED-DATA.indication	
TRECVERR		
TRECVFR	T-UNITDATA.indication	
TREJECT	T-DISCONNECT.request	
TRELACK	T-RELEASE.indication	
TRELEASE	T-RELEASE.request	
TRETRACT		
TSEND	T-DATA.request or T-EXPEDITED-DATA.request	2
TSENDTO	T-UNITDATA.request	
TSTATE		
TUNBIND		
TUSER		

Note: These notes correspond to the note numbers in the table:

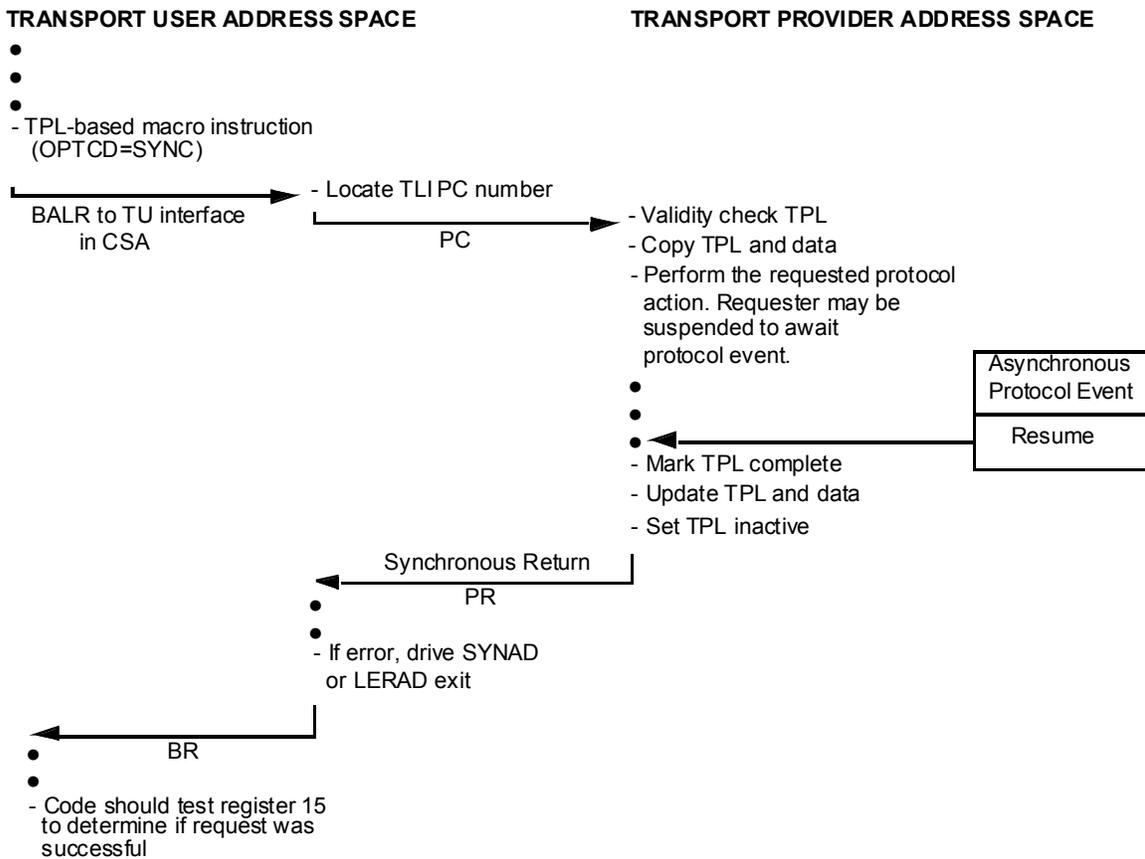
1. The API issues a synchronous TCLOSE for all opened endpoints.
2. Primitive issued to a TCP-based provider may delay completion until an acknowledgment is received from the remote transport provider.
3. The API may issue a system WAIT macro instruction if pending request has not been completed.
4. The API issues T-DISCONNECT.request primitive if connection has not been released.

Synchronous Operation

In a synchronous program, operations are performed serially. A request for synchronous operation (OPTCD=SYNC) means that the API does not return control to the next sequential instruction in the application program task from which the macro instruction was issued until after the requested operation completes. Execution of the application program task is suspended by issuing a system WAIT macro instruction until the API completes the request. The program must wait for the processing of one requested operation to complete before going on to the next.

Synchronous Operation Flow

The following diagram shows the flow of a synchronous operation.



When control is returned to the next sequential instruction in the application program task, the TPL is set inactive, and may be reused for another request. Synchronous mode is appropriate for application programs that manage a single endpoint and do not need to overlap endpoint operations, such as send and receive operations, with other application program activities. This suggests that the communication of data via the endpoint is half-duplex in nature, and that some arrangement exists between the communicating entities to know when to send and receive data. Since only one operation may be pending at any point in time, synchronous operation is usually less complex than asynchronous operation.

While the application program is waiting for the request to complete, an asynchronous event such as a timer interrupt could cause the program's `STIMER` exit routine to be entered. Similarly, asynchronous API events could cause the corresponding exit routines to be entered (see the discussion of exit routines in [Specifying and Using Exit Routines](#) . Only the application program task from which the macro instruction was issued is suspended while waiting for completion of a synchronous operation. The exit routines associated with the program are scheduled and executed whether or not the mainline program logic is awaiting completion of a synchronous operation. However, if a synchronous operation is issued from within an exit routine, executions of other exit routines may be prevented until the operation completes.

When a synchronous operation completes, the application program must determine whether the operation was successful or unsuccessful. The program does this by testing values in registers 15 and zero and by examining fields in the TPL used for the operation. For more information on error handling, refer to [Handling Errors and Special Conditions](#) .

Asynchronous Operation

In an asynchronous operation, control of execution may return to the application program's next sequential instruction before the requested operation completes. The transport provider may not be able to complete an operation immediately if dependent upon some event, such as the arrival of data or a protocol indication, or if it is necessary to serialize the request with other API requests for a given endpoint. In these situations, the transport provider analyzes request for errors and saves the request to await the respective event before returning to the application program. At this point, the program may issue other API requests or perform other application specific activities.

Example

An application program can issue a `TRECV` macro instruction on one endpoint indicating asynchronous mode, and while the input operation is being performed, the application program can:

- Send some data on the same endpoint
- Initiate an operation on a different endpoint

- Perform other I/O activities, such as reading or writing a direct-access storage device

The application program can determine if the request was accepted by testing the value returned in register 15. If the value is zero, the request was accepted and the associated TPL is set active. If the value is non-zero, the request was not accepted, and register 15 and zero contain the same information as if the request was executed in synchronous mode. In this case, the TPL sets inactive, and the application program should initiate error recovery without waiting for the request to complete.

While an asynchronous operation is pending, the associated TPL is active and cannot be used with another request until it is posted complete. If the application program issues another TPL-based macro instruction, a different TPL must be used. In addition, when an operation is pending for an endpoint, other operations that can be initiated at the same endpoint are limited.

When an asynchronous operation is specified, there are two ways the API can notify the application program that the requested operation has completed:

- If the application program associates an Event Control Block (ECB) with the request, the API posts the ECB when the operation completes.
- Alternatively, the application program can designate that a particular TPL exit routine be executed as soon as the operation completes.

When the operation completes, the API schedules the exit routine.

The method of notification is controlled by storing the address of the ECB or exit routine in the TPL used for the request.

Regardless of whether a program waits on an ECB or uses a TPL exit routine, a TCHECK macro instruction must be executed after an asynchronous operation completes to set the TPL inactive and to make it available for another request. The TCHECK macro instruction also clears the ECB if one was provided.

It is also important to note that the TPL ECB may be posted or the TPL exit routine may be executed before the application receives control back from the API (for example, at the next sequential instruction following the TLI request macro). This is because this activity occurs upon a different unit of execution within the operating system.

Asynchronous Operation Using ECBs

By using ECBs, the application program can issue one WAIT macro instruction for a combination of pending API requests in addition to non-API requests that also use ECBs.

Example 1

An application program can issue three TRECVR requests for three different endpoints and three BSAM WRITE requests for three different data sets. By issuing one WAIT for all six ECBs, the application program resumes processing when any one of the six operations completes. When execution resumes, the application program can determine which operation completed by determining which ECBs are posted.

The distinction between ECBs and TPL exit routines is primarily that the TPL exit routine is automatically scheduled when the requested operation completes, thereby saving the application program the trouble of testing ECBs and branching to subroutines. On the other hand, the use of ECBs provides the program with greater control over the order in which events are to be handled.

Example 2

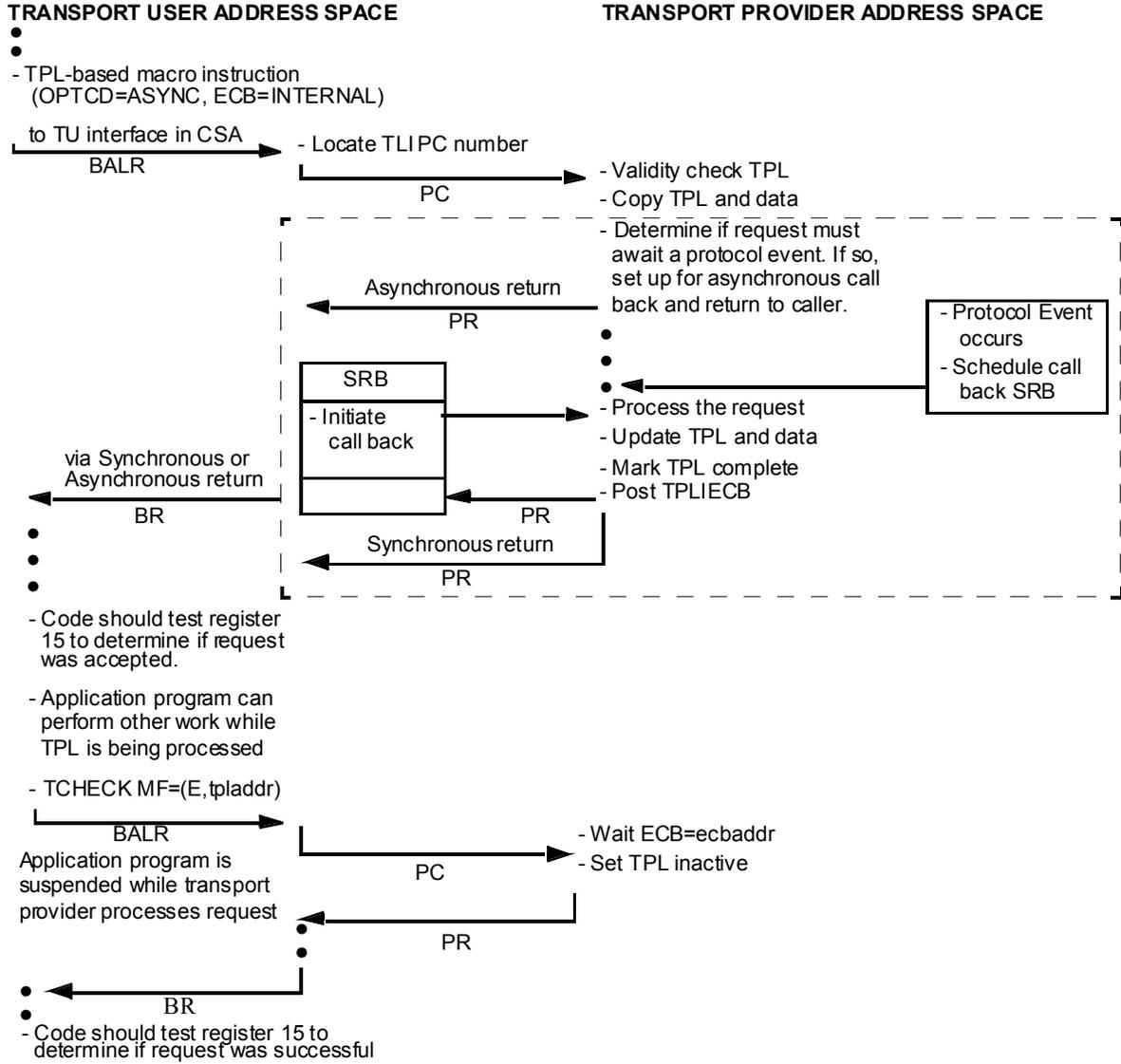
The application program can prioritize requested operations by testing some ECBs before others. The order of testing can be varied during program execution as circumstances change.

If neither an ECB address nor a TPL exit routine address is specified by the TPL-based macro instruction, the API uses the ECB-EXIT field of the TPL (TPLECBXR) as an internal ECB, and the API (for synchronous operations) or the application program (for asynchronous operations) waits on it, checks and clears it. Alternatively, the ECB-EXIT field can be set to point to an external ECB provided by the application program by using a TPL-based macro instruction that specifies ECB=ecb address. Once set, it can be reset to an internal ECB by specifying ECB=INTERNAL. If OPTCD=SYNC is specified, ECB=INTERNAL is assumed.

Asynchronous Operation Using Internal ECB

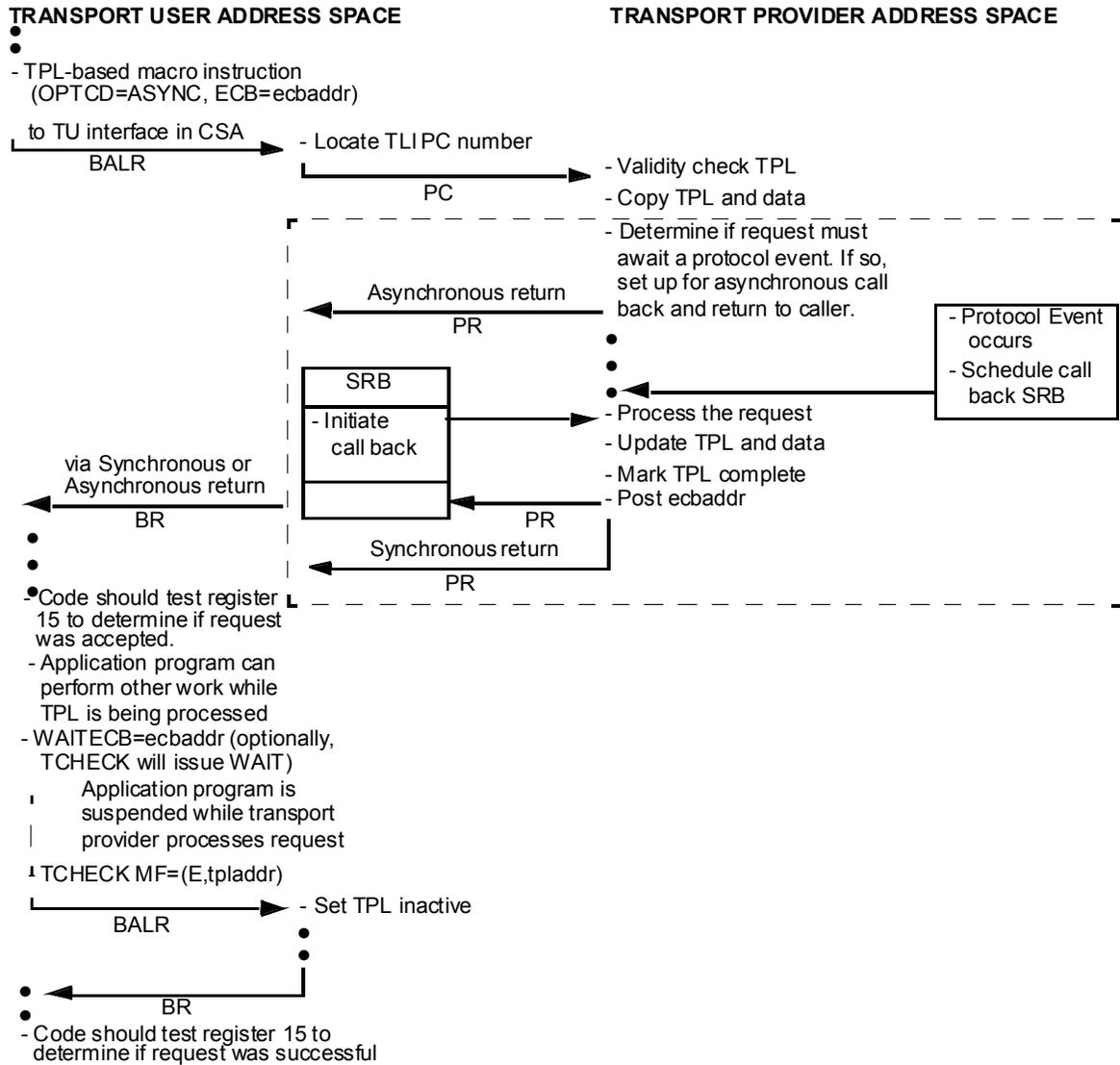
When using an internal ECB, the application program does not wait or check the internal ECB, and lets the TCHECK function wait if necessary. The application program could have waited on the ECB explicitly knowing that it is located in the TPL location at TPLIECB (TPLECBXR).

The diagram, Asynchronous Operation Using External ECB, shows this processing flow.



Asynchronous
Operation Using
External ECB

When using an external ECB, the application program issues a WAIT for the external ECB, but could have allowed TCHECK to perform the wait implicitly. This diagram shows this processing flow.



Asynchronous
Operation Using
Completion Exits

Instead of having the API post an ECB when a request for an asynchronous operation completes, the application program can have the API schedule and cause control to be given to a TPL-specified asynchronous exit routine. The TPL exit routine can supply the logic that would have been branched to by the mainline program after discovering a posted ECB. A TPL exit routine is any exit routine whose symbolic name was provided in the EXIT operand of the macro instruction or the TPL used for the request.

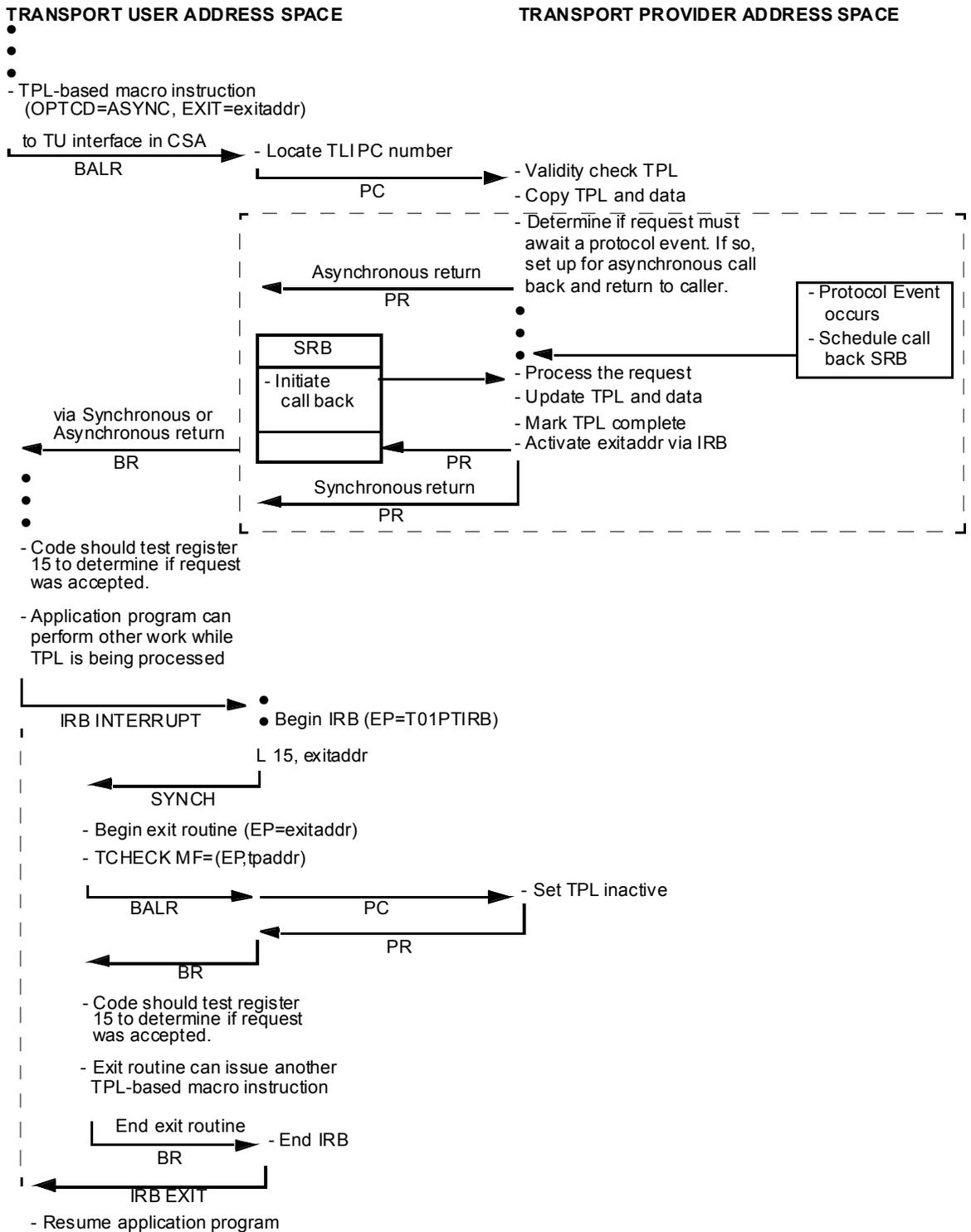
One advantage of using a TPL exit routine instead of an ECB is that it is easier to code for this type of processing than it is to code the logic associated with discovering a posted ECB and relating the ECB to a branch address. Also, the TPL exit routine is given control almost immediately after the associated TPL-based operation completes, and thus has priority over the mainline program. A disadvantage of a TPL exit routine is that more system instructions must be executed to schedule an exit routine than must be executed to post an ECB. Also, as discussed in the previous section, TPL-exit scheduling does not provide as much flexibility as ECB-posting for giving a higher priority to selected operations. An application program can use a combination of ECB-posting and TPL exit routines (see [Mixing Synchronization Modes](#) on mixing modes of operation).

A TPL exit routine can itself issue asynchronous requests, continue executing, and return to the API. The asynchronous request in a TPL exit routine can specify that on completion of the request, an ECB be posted or a TPL exit routine be scheduled. If the TPL exit routine option is chosen, the exit routine can be the same one in which the request was issued, or different.

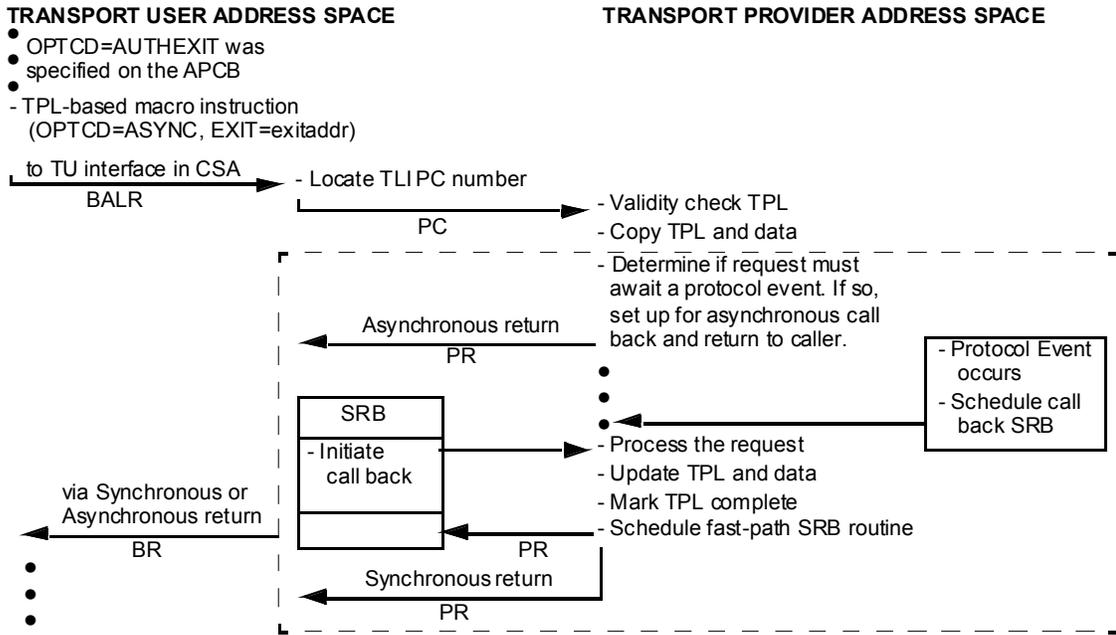
A TPL exit routine can also issue synchronous requests, but the exit routine is suspended until the synchronous operation completes. Since the API schedules exit routines serially for a given task, a synchronous request issued in an exit routine also prevents other exit routines from receiving control until control is returned from the suspended exit routine. The TPL exit routine may have run, or the TPL completion ECB may have been posted prior to the return from the asynchronous request.

The following diagram, Asynchronous Operation Using Completion Exits, shows this processing flow.

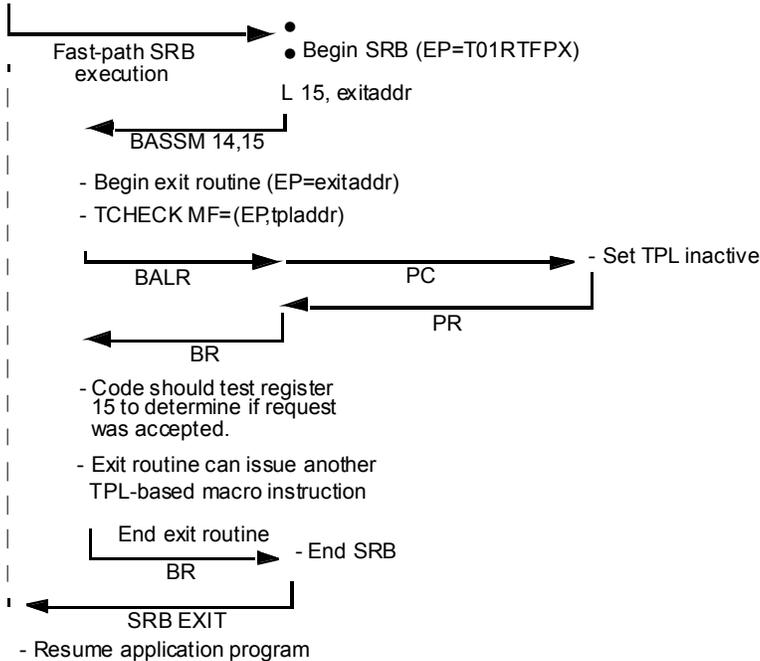
Asynchronous Operation Using Completion Exits



Asynchronous Operation Using Authorized Completion Exits



- Code should test register 15 to determine if request was accepted.
- Application program can perform other work while TPL is being processed



Using the TCHECK Macro Instruction

When an asynchronous request is accepted by the API, control is returned to the next sequential instruction of the application program with the TPL marked as active. No other request can be issued with an active TPL until the operation completes and the TPL is marked inactive.

A TCHECK macro instruction must be executed to set the TPL inactive. If ECB-posting was specified for a request, the application program generally waits for the ECB to post and then issues the TCHECK macro instruction. However, if a TCHECK macro instruction executes before the ECB is posted, the API waits for the ECB to post. In either case, the ECB is cleared by the API before returning to the application program.

When a TPL exit routine is specified, the TCHECK macro instruction is usually executed by the exit routine. However, the TCHECK macro instruction can be executed by the mainline program after the exit routine has run, but this may be difficult to coordinate. If the TCHECK macro instruction executes before the operation completes (and EXIT is specified), an error is generated.

The TCHECK macro instruction also causes error recovery exits to be entered. This is covered in more detail in [Handling Errors and Special Conditions](#).

Mixing Synchronization Modes

Since the synchronization mode used for a request is determined at the time the request is issued, the modes of operation previously described may be mixed. That is, an application program can issue some requests in synchronous mode, and issue others in asynchronous mode. Similarly, some asynchronous requests can specify ECB-posting, and others can specify TPL exit routines.

The important point to remember is that once a synchronous request is issued, the operation must complete before another request can be issued from the same program thread. If the program thread is the mainline program, exit routines can still run and requests can be issued from those exit routines. However, if a synchronous request is issued by an exit routine, no other API-scheduled exit routines run until the synchronous operation is complete. This fact can be used to the advantage or detriment of the application program.

Specifying and Using Exit Routines

The API lets an application program use exit routines to gain control to handle certain events. An exit routine is written to handle a specific event, such as, a SYNAD routine is written to process TPL-based errors or special conditions other than program logic errors). When the event occurs, the API gives the exit routine control as soon as possible.

This section discusses:

- How exit routines are specified
- How they are called by the API
- Describes procedures to follow in using them

An overview of the API exit routines is presented, and then each exit routine is described in detail.

How Exit Routines Are Specified

The API provides for the use of two general kinds of exit routines by an application program:

- TPL request completion exit routines
- TEXTLST or TEVNTLST protocol event exit routines

These two types of exit routines work somewhat differently and derive their names from how they are specified.

TPL Exit Routines

The instructions to execute when a TPL-based service request completes can be written as a separate routine. This routine, called a TPL exit routine, can be specified in the TPL-based macro instruction used to initiate the request.

The address of the exit routine is specified by the EXIT operand of the macro instruction or is stored in the TPLEXIT field of the TPL. When the requested operation completes, the API schedules and eventually causes entry to the TPL exit routine. The same TPL exit routine can be designated by more than one TPL-based macro instruction. That is, a TPL exit routine can be established as a common exit routine.

TEXTLST and TEVNTLST Exit Routines

TEXTLST and TEVNTLST exit routines differ from TPL exit routines in being special-purpose. That is TEXTLST/TEVNTLST exit routines handle special events that are well understood by both the application program and the API. Instead of being specified in a particular TPL-based macro instruction request, the identity of a TEXTLST/TEVNTLST exit routine is established only when the exit list in which its name is specified is identified to the API, either when the application program session is established with the API or, for certain types of exit routines, when an endpoint is opened. In other words, TPL exit routines are associated with a particular service request, and TEXTLST/TEVNTLST exit routines are associated with a transport user or a particular endpoint.

The TEXTLST macro instruction is used to build an exit list, and the exit list is associated with the transport user by linking it to the APCB specified in the AOPEN macro instruction. For those exits that can be associated with a particular endpoint, the exit list must be specified in the TOPEN macro instruction. The TEXTLST macro instruction keyword used to specify the exit routine's address is used in the remainder of this section when discussing particular types of exit routines.

Example

The exit routine specified by the TPEND keyword is referred to as the TPEND exit routine.

Note: The TEVNTLST macro instruction provides capabilities similar to TEXTLST. TEVNTLST, however, can only be used with the TOPEN macro instruction. TEVNTLST may not be referenced by an APCB macro instruction.

Internally, the TEVNTLST macro generates a parameter list that is identical to the parameter list generated by TEXTLST. TEVNTLST provides the additional capability of specifying ECB addresses in lieu of exit routines. Both ECB addresses and exit routine addresses can be used in a TEVNTLST specification, but for a given event, only one ECB or exit address can be specified.

When specifying TEXTLST exit routines, the content and length of an exit list is dependent on whether the exit list is used with an AOPEN or TOPEN macro instruction, and must be indicated with the TEXTLST macro instruction. An AOPEN exit list cannot be used with TOPEN, and vice versa.

The AOPEN and TOPEN event lists are hierarchical. When a particular endpoint event occurs that is handled by the application program, the TOPEN event list associated with the endpoint is accessed first. If no exit routine or ECB address was specified in the TOPEN event list, then the AOPEN exit list is used. Only certain types of exit routines can be specified in a TOPEN event list. However, all types can be specified in an AOPEN exit list. Thus, the AOPEN exit list can be used to provide default exit routines for the application program in general, and the TOPEN exit list can provide special exit routines and ECBs for particular endpoints if and when the need arises.

How Exit Routines Are Called

Exit routines are classified as synchronous or asynchronous exit routines according to how they are called by the API. The SYNAD and LERAD exit routines are synchronous, and all other exit routines are asynchronous. Exit routines have different characteristics based on their classification. This classification should not be confused with similar terminology used to describe modes of operation for service requests.

Synchronous Exit Routines

Synchronous exit routines are sometimes referred to as inline exit routines and are considered to be an extension of the part of the application program (either mainline or asynchronous exit routine) that was executing when the synchronous exit routine was invoked. Effectively a synchronous exit routine is a branch entered from a TPL-based or TCHECK macro instruction just before control is returned to the next sequential instruction in the application program.

After a synchronous exit routine completes processing, it can return to the API. If it is coded to return, the application program receives control at the next sequential instruction immediately after the TPL-based or TCHECK macro instruction that caused the synchronous exit routine to be invoked. If the exit routine is coded not to return, it is as if the application program branched to another location after issuing the TPL-based or TCHECK macro instruction.

Register Information

When a synchronous exit routine is entered, the general-purpose registers contain the following information:

- R0 Recovery action code.
- R1 Address of TPL.
- R2-12 Unmodified application program registers.
- R13 Unmodified application program save area address.
- R14 API return address.
- R15 Address of exit routine.

Synchronous Exit
Routine Coding
Procedures

These procedures should be followed when coding a synchronous exit routine:

- The exit routine is not obligated to return to the API. However, if it does, register 14 should be saved and restored before returning.
- The exit routine is not required to save and restore general-purpose registers 2-12, and the API does not provide a save area for the exit routine. However, register 13 does contain the address of an 18-word save area supplied by the application program.
- If the exit routine issues any TPL-based macro instructions or calls any external routines, it must provide the address of its own 18-word save area in register 13. Register 13 should be restored before returning to the API.
- If the exit routine returns to the API, it should not execute any macro instruction or call any routine that would change the contents of the 18-word save area supplied by the application program. The API restores the application program's registers from the save area before returning to the next sequential instruction.
- Registers zero and 15 are not restored by the API and the exit routine is not required to preserve them. Contents of these registers are passed to the application program if the exit routine returns to the API.

A synchronous exit routine is generally not required to be reentrant. However, if a synchronous exit routine executes other TPL-based macro instructions, the exit routine can be reentered (recursion). In addition, if TPL-based service requests are issued from the mainline program and asynchronous exit routines, or the synchronous exit routine is shared by two or more programs, it can be reentered. In this case, the application program should be prepared to handle reentrancy issues (for example, dynamic allocation of work areas).

Here is an example of a synchronous exit routine that returns to the API. An 18-word save area is provided since it may issue other service requests or call external routines:

```
*****
*   SYNCHRONOUS ERROR RECOVERY ROUTINE
*****
        USING SYNADX,12
SYNADX  LR   12,15          ESTABLISH BASE ADDRESS
        LR   2,0           SAVE RECOVERY ACTION CODE
        LTR  3,1           TEST TPL ADDRESS
        BNP  RECURSIV     IGNORE IF CALLED RECURSIVELY
        LR   5,14         OTHERWISE, SAVE RETURN ADDRESS
        O    3,=X'80000000' SET RECURSION FLAG
        ST   13,SYNADSAV+4 BACK CHAIN SAVE AREAS
        LA   13,SYNADSAV  ESTABLISH NEW SAVE AREA
        .
        . [this exit routine is not intended to be reentrant]
        .
        TERROR VERBATIM,MF=(E,(3)) GENERATE DIAGNOSTIC MESSAGE
        LTR  15,15         MESSAGE RETURNED?
        BNZ  SKIPWTO      IF NOT, SKIP WTO
        LR   4,0           SAVE WTO LIST ADDRESS
        XR   0,0
        USING TEM,4
        WTO  MF=(E,TEMWTO) WRITE MESSAGE TO OPERATOR
        L    0,TEMSL      LOAD LENGTH AND SUBPOOL
        FREEMAIN R,LV=(0),A=(4)
        RELEASE MESSAGE STORAGE AREA
        DROP 4
SKIPWTO DS   0H
        .
        . [perform recovery processing]
        .
RETURN   L    13,4(,13)    RESTORE PREVIOUS SAVE AREA ADDRESS
        LR   14,5         RESTORE RETURN ADDRESS
        XR   15,15        UPDATE GENERAL RETURN CODE
        XR   0,0         CLEAR CONDITIONAL COMPLETION CODE
RECURSIV DS   0H
        BR   14           RETURN TO THE API
        LTORG
        .
        .
SYNADSAV DS   18F        EXIT ROUTINE SAVE AREA
        TDSECT TEM       GENERATE TEM DSECT
```

Asynchronous Exit Routines

Asynchronous exit routines, in contrast to synchronous exit routines, do not act as extensions to the part of the application program that was executing when the event associated with the exit routine occurred. The events that cause invocation of asynchronous exit routines are unpredictable, whereas synchronous exit routines can be invoked only at predictable points, for instance, immediately after the associated TCHECK or TPL-based macro instruction.

Asynchronous exit routines can interrupt the mainline program at any time, even if the mainline program is currently suspended (for example, because it issued a TCHECK or WAIT macro instruction). However, no API-invoked asynchronous exit routine can interrupt another asynchronous exit routine; thus each asynchronous exit routine must return to the API before the next asynchronous exit routine can be given control.

When an asynchronous event occurs, the associated exit routine (if defined by the application program) is scheduled by the API by being put on a transport user exit queue. If the mainline program is currently in control, execution of the mainline program is suspended and control is immediately given to the exit routine. If another asynchronous exit routine is in control, that exit routine must return to the API before the next exit routine on the user exit queue can be given control. If the asynchronous exit routine currently in control suspends execution (for example, by issuing TCHECK or WAIT), it prevents other asynchronous exit routines from gaining control. When the final asynchronous exit routine returns to the API (the transport user exit queue is now empty), the mainline program resumes control at the point where it was interrupted.

The API schedules exits only for the API events, and is unaware of asynchronous exits scheduled by other program products or the operating system. Therefore, the serialization of asynchronous exits described here only applies to the API exits, and only applies to exits for one transport user (that is, task).

Example

The API asynchronous exits can be interrupted by VTAM exits, and VTAM asynchronous exits can be interrupted by the API exits. Additionally, the API and VTAM exits can be interrupted by asynchronous exits scheduled by MVS (for example, STIMER exits).

The application program must implement the appropriate serialization mechanisms when common data areas can be accessed by different asynchronous exit routines that are interruptible by one another.

Register Information When an asynchronous exit routine is entered, the general-purpose registers contain this information:

- R0 Unpredictable.
- R1 Address of TPL or TXP.
- R2-12 Unpredictable.
- R13 Unpredictable.
- R14 API return address.
- R15 Address of exit routine.

Asynchronous Exit
Routine Coding
Procedures

These procedures should be followed when coding an asynchronous exit routine:

- Asynchronous exit routines must return control with a BR 14 after register 14 is restored with the address it contained when the exit routine was entered.
- Except for general-register 14, the exit routine is not required to save the API registers; register 13 does not contain the address of an API or application program save area.
- If the exit routine issues any TPL-based macro instructions or calls any external routines, it must provide the address of its own 18-word save area in register 13.
- The AOPEN or ACLOSE macro instruction cannot be executed in an asynchronous exit routine (the API or otherwise). The AOPEN or ACLOSE macro instruction must always be executed from the mainline program.
- Care must be taken not to reuse save areas still in use by other parts of the application program. For example, the save area used by the mainline program when a TPL-based macro instruction is issued is in use until the API returns to the mainline program. It should not be used by an asynchronous exit routine in the meantime.
- If the exit routine issues a TPL-based macro instruction and completion is awaited in the same routine (for example, by the use of TCHECK or WAIT) the mainline program, as well as the exit routine, is suspended until the requested operation completes. To avoid such delays, consider using a TPL exit routine for notification of completion.

Since asynchronous exit routines are serialized by the API, they are not normally reentered. However, if the same asynchronous exit routine is shared by two or more tasks, it must be coded to be reentrant. In this case, save areas and work areas should be allocated dynamically.

Asynchronous Exit
Routine Example

Here is an example of an asynchronous exit routine. In this example, the exit routine is entered when data is available to be received. The exit routine issues a TRECVC macro instruction in asynchronous mode and returns to the API. The TPL completion exit is also shown.

```
*****
* ASYNCHRONOUS EXIT ROUTINE TO HANDLE DATA INDICATIONS
*****
        USING DORECV,12
DORECV  LR   12,15          ESTABLISH BASE ADDRESS
        LR   2,1           MOVE TXP ADDRESS
        LR   3,14          SAVE RETURN ADDRESS
        LA   13,SAVEAREA   ESTABLISH SAVE AREA
        USING TXP,2
        L    4,TXPEPID     LOAD ENDPOINT ID
        LA   5,RECVTPL     LOAD RECEIVE TPL ADDRESS
        USING TPL,5
        TRECVC EP=(4),MF=(E,(5))  INITIATE RECEIVE REQUEST
        LTR  15,15         REQUEST ACCEPTED?
        BZ   RECVEXIT     IF SO, RETURN TO API
RECVERR DS   OH
        .
        . [perform error recovery processing]
        .
        DROP 2
        USING RECVDONE,15
*****
* TPL EXIT ROUTINE FOR TRECVC SERVICE REQUEST
*****
RECVDONE L   12,=A(DORECV)  ESTABLISH COMMON BASE ADDRESS
        DROP 15
        LR   5,1           MOVE TPL ADDRESS
        LR   3,14          SAVE RETURN ADDRESS
        LA   13,SAVEAREA   ESTABLISH SAVE AREA
        TCHECKMF=(E,(1))   CHECK TPL AND SET INACTIVE
        LTR  15,15         RECEIVE SUCCESSFUL?
        BNZ  RECVERR     IF NOT, GO HANDLE ERROR
        .
        . [received data is now available for processing]
        .
RECVEXIT LR   14,3         RESTORE RETURN ADDRESS
        BR   14           RETURN TO API
        DROP 5,12
        LTORG
        .
        .
RECVTPL  TRECVC DABUF=DATAAREA,DALEN=L'DATAAREA,          +
        OPTCD=ASYN,EXIT=RECVDONE,MF=L
*
DATAAREA DS   XL1024     RECEIVE DATA BUFFER
SAVEAREA DS   18F       EXIT ROUTINE SAVE AREA
        TDSECT TXP,TPL   GENERATE DSECTS
```

Exit Routine Parameter List

The value passed to an exit routine in register one is always the address of a parameter list. For the SYNAD, LERAD, and TPL exit routines, this parameter list is the TPL, which is in error or has just completed. For all other exit routines, a common API data structure is used to pass exit routine parameters. This data structure is defined by the TXP DSECT. Refer to the *Unicenter TCPaccess Communications Server Assembler API Macro Reference* for more information.

Transport Endpoint
Exit Parameters (TXP)

The following table shows the transport endpoint exit parameters (TXP).

TXP+0	TXTYPE	Reserved
+4	TXPEPID	
+8	TXPEXIT	
+12	TXPPARM	
+16	TXPACNTX	
+20	TXPUCNTX	
+24	TXPECNTX	
+28	TXPPARM2	

TXP Information

The TXP DSECT can be generated by the TDSECT macro instruction.

The following describes the contents of the TXP.

TXPTYPE	A halfword integer value indicating the type of exit. The exit type defines the contents of the TXPPARM field.
TXPEPID	Identifies the endpoint associated with the event. If there is no endpoint associated with the event (for example, TPEND exit), the value of this field is zero.
TXPEXIT	Contains the entry point address of the exit routine. The API uses this field to schedule the exit routine.
TXPPARM	Contains a fullword parameter that is exit-dependent. For protocol event exits, the parameter identifies a particular protocol event. For the TPEND exit, this field contains a reason code.
TXPACNTX	Contains a word of context associated with the application program. This value is copied from the APCB and is specified by the ACNTX keyword on the APCB macro instruction.

TXPUCNTX	Contains a word of context associated with the endpoint. This value is copied from an internal data structure and is equal to the value specified by the UCNTX keyword on the TOPEN macro instruction.
TXPECNTX	Contains a word of context associated with the language environment. If the language environment is assembler language, this value is equal to the value specified by the ECNTX keyword on the APCB macro instruction.
TXPPARM2	Contains additional exit-dependent information. In the case of data indication, it is the amount of available window space.

The TXP is dynamically formatted in key-0 storage and cannot be modified by an exit routine executing with its normal private-area key.

How Exit Routines Are Used

This section summarizes all API exit routines and provides an exit routine register usage summary. Usage of each exit routine is also discussed.

Exit Routine Summary

This table lists each API exit, the type of exit, how and when it is specified, and the purpose for which it is used.

Exit	Type	How Specified*	When Specified	Purpose
CONFIRM	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to receive a confirm indication. Exit routine should respond by issuing a TCONFIRM macro instruction.
CONNECT	Asynchronous	TEXLST or TEVNTLST	AOPEN or TOPEN	Used to receive a connect indication. Exit routine should respond by issuing a TLISTEN macro instruction.

Exit	Type	How Specified*	When Specified	Purpose
DATA	Asynchronous	TEXTLST or TEVNTLST	AOPEN or TOPEN	Used to receive a normal data (COTS) or datagram (CLTS) indication. Exit routine should respond by issuing a TRECVR or TRECVRFR macro instruction.
DGERR	Asynchronous	TEXTLST or TEVNTLST	AOPEN or TOPEN	Used to receive a datagram error indication. Exit routine should respond by issuing a TRECVRERR macro instruction.
DISCONN	Asynchronous	TEXTLST or TEVNTLST	AOPEN or TOPEN	Used to receive a disconnect indication. Exit routine should respond by issuing a TCLEAR macro instruction.
LERAD	Synchronous	TEXTLST	AOPEN only	Used to recover from program logic errors that typically occur during program debugging.
RELEASE	Asynchronous	TEXTLST or TEVNTLST	AOPEN or TOPEN	Used to receive a release indication. Exit routine should respond by issuing a TRELACK macro instruction.
SENDWIND	Asynchronous	TEXTLST or TEVNTLST	AOPEN or TOPEN	Used to notify MODE=SOCKETS endpoints that the send window has opened.
SYNAD	Synchronous	TEXTLST	AOPEN only	Used to recover from physical errors and exceptional conditions.
TPEND	Asynchronous	TEXTLST or TEVNTLST	AOPEN or TOPEN	Called when the transport provider or the API is stopped or terminated. Exit routine should initiate shutdown procedures.
APEND	Asynchronous	TEXTLST	AOPEN only	Called when the API is stopped or terminated. Exit routine should initiate shutdown procedures.

Exit	Type	How Specified*	When Specified	Purpose
TPL	Asynchronous	EXIT operand on TPL-based macro instruction	Any TPL- based service request	Called when the requested operation is complete. Exit routine should issue TCHECK macro instruction to set TPL inactive.
XDATA	Asynchronous	TEXTLST or TEVNTLST	AOPEN or TOPEN	Used to receive an expedited data indication. Exit routine should respond by issuing a TRECV macro instruction.

* Any exit that can be specified on a TOPEN TEXTLST can be specified on a TOPEN TEVNTLST as well.

Register Usage Summary

The following table summarizes the register usage for each type of exit routine.

Contents Of General-Purpose Registers On Entry To Exit Routine						
Exit	Register 0	Register 1	Register 2-12	Register 13	Register 14	Register 15
CONFIRM	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
CONNECT	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
DATA	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
DGERR	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
DISCONN	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
LERAD	Recovery action code	Address of TPL	Unmodified	Unmodified	API return address	Address of exit routine
RELEASE	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
SENDWIND	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
SYNAD	Recovery action code	Address of TPL	Unmodified	Unmodified	API return address	Address of exit routine

Contents Of General-Purpose Registers On Entry To Exit Routine						
Exit	Register 0	Register 1	Register 2-12	Register 13	Register 14	Register 15
TPEND	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
APEND	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine
TPL	Unpredictable	Address of TPL	Unpredictable	Unpredictable	API return address	Address of exit routine
XDATA	Unpredictable	Address of TXP	Unpredictable	Unpredictable	API return address	Address of exit routine

Note: For APEND exits, the TXPUCNTX field is not set; the TXPAPCB field is set.

TPL Completion Exit

A TPL exit routine is entered after a TPL-based operation completes if the TPL or the macro instruction using it specified the exit routine address in the EXIT operand. Specifying an exit routine forces the synchronization mode to be asynchronous (OPTCD=ASYNCR).

The TPL exit routine is entered with the address of the TPL in register one. The TPL is marked complete, but remains active until a TCHECK macro instruction is executed. The TCHECK macro instruction can be issued by the TPL exit routine, or later by posting an ECB on which the mainline program is waiting. If the TCHECK macro instruction is issued before the TPL exit routine is entered, TCHECK returns with an error.

If the application program identifies a SYNAD or LERAD exit routine in the exit list provided to AOPEN, the appropriate exit routine is entered if the TPL service request completed with an error. Otherwise, the TPL exit routine should check register 15 returned by the TCHECK macro instruction, or check the return code fields of the TPL to determine if the request completed successfully.

The exit routine can initiate additional API service requests using the same TPL, or can supply a different TPL. If additional TPL-based requests are executed, it is best that the exit routine not wait for completion, otherwise entry into other asynchronous exit routines may be delayed. Newly initiated requests should specify posting of an ECB or a TPL exit routine, which in the latter case, may be the same exit routine. When processing completes, control must be returned to the API by branching to the address contained in register 14 when the exit routine was entered.

Protocol Event Exits and ECBs

Certain protocol events can be handled asynchronously by the application program. These protocol events generally correspond to T.indication or T.confirm primitives issued by the transport provider.

All protocol event exits are entered with register 1 containing the address of a TXP. The TXP identifies the exit type (TXPTYPE) as a protocol event (TXPTPROT), TXPEPID contains the endpoint ID for the endpoint associated with the protocol event, and TXPPARM contains a fullword value identifying the particular event. The event code is in multiples of four and can be used as a branch or table index to locate the appropriate processing routine.

The following protocol event codes are defined:

Name	Dec	Hex	Exit	Protocol Event
TXPECONN	0	X'00'	CONNECT	Connect indication.
TXPECONF	4	X'04'	CONFIRM	Confirm indication.
TXPEDATA	8	X'08'	DATA	Normal data (or datagram) indication.
TXPEXPDT	12	X'0C'	XDATA	Expedited data indication.
TXPERROR	16	X'10'	DGERR	Datagram error indication.
TXPEDISC	20	X'14'	DISCONN	Disconnect indication.
TXPERLSE	24	X'18'	RELEASE	Orderly release indication.
TXPESWND	28	1C	SENDWIND	Send Window Open.

The event code is used by the API to determine which exit routine address to load from the exit list. Therefore, the application program can supply a different exit routine for each protocol event, or supply a common exit routine that handles all events, or a particular subset of events. If no exit routine is specified in the TOPEN or AOPEN exit list, the application program cannot receive asynchronous notification of the corresponding protocol event.

A protocol event exit routine generally issues an appropriate TPL-based macro instruction to clear the pending indication. Since the indication serves to notify the application program that some awaited event has occurred, it is generally safe to issue the responding macro instruction in synchronous mode without fear of suspending the exit routine for an indefinite period. However, the exit routine must be suspended long enough for the API address space to complete the service request.

Note: For optimum performance, it may be better to execute the request in asynchronous mode and supply a TPL exit routine to handle the subsequent completion.

The exit routine is not obliged to clear the pending indication in the exit routine. Instead, an ECB could be posted on which the mainline program is waiting. When the exit routine completes its processing, control must be returned to the API by branching to the address contained in register 14 when the exit routine was entered.

CONNECT Event

The CONNECT exit routine is scheduled or the ECB is posted when a connect indication is generated and no TLISTEN service request is outstanding. That is, the TLISTEN request is issued and is awaiting completion. A connect indication is generated by the API on receiving a T-CONNECT.indication primitive issued by the transport provider. Connect indications can only be generated on endpoints in the enabled (TSENABLD) or connect-indication-pending (TSINCONN) state.

The CONNECT exit routine should execute a TLISTEN macro instruction to receive the connect indication. Once a connect indication is received with TLISTEN, it is said to be pending until accepted or rejected with a TACCEPT or TREJECT macro instruction. The exit routine is not obliged to issue the TLISTEN request, and may defer this responsibility to the mainline program by posting an ECB. Similarly, the TACCEPT or TREJECT macro instruction may be issued by the mainline program or in a subsequent TPL completion exit to prevent suspension of the CONNECT exit routine.

Once the CONNECT exit routine is scheduled, it is not rescheduled until all available connect indications are received by a TLISTEN macro instruction. If another T-CONNECT.indication is issued by the transport provider, it is queued without rescheduling the CONNECT exit routine. Therefore, once the CONNECT exit routine is entered, sufficient TLISTEN service requests should be issued to receive all of the indications that were queued. The COUNT field (TPLCOUNT) returned in the TPL by TLISTEN indicates the number of additional connect indications to be received. Also, a bit (TOMORE) is set in the option code field (TPLOPCD2) to indicate more connect indications are available. The application program should receive and accept or reject connect indications quickly since a limited number may be queued (see the QLSTN parameter on the TBIND macro instruction in the *Unicenter TCPaccess Communications Server Assembler API Macro Reference*).

When associations are used with an endpoint operating in connectionless mode (CLTS), datagrams arriving from a new source address cause a connect indication to be simulated. The datagram is queued until the connect indication is received and accepted. Thus, the CONNECT exit routine can be used to form CLTS associations in the same manner it is used to establish COTS connections.

CONFIRM Event

The CONFIRM exit routine is scheduled when a confirm indication is generated and no TCONFIRM service request is outstanding. A confirm indication is generated by the API on receiving a T-CONNECT.confirm primitive issued by the transport provider.

Confirm indications can only be generated on endpoints in the connect-in-progress (TSOUCONN) state. That is, a TCONNECT macro instruction has successfully executed at the endpoint and the application program is awaiting connect confirmation).

The CONFIRM exit routine should execute a TCONFIRM macro instruction to receive the confirm indication.

If the TCONFIRM macro instruction completes successfully, the connection is established and the state of the endpoint becomes connected (TSCONNECT).

If the application program no longer desires to establish the connection, a TDISCONN macro instruction should be executed instead.

The exit routine is not obliged to receive (or clear) the confirm indication and can defer this responsibility to the mainline program by posting an ECB on which it is waiting.

A confirm indication can only occur as the result of a successful TCONNECT service request. Therefore, only one confirm indication can be received on a given endpoint until the connection is released, and another TCONNECT request is issued. If the endpoint is operating in connectionless mode with associations, a confirm indication is simulated when the TCONNECT request is issued. Thus, the CONFIRM exit routine can be used to form CLTS associations in a manner similar to COTS connections.

DATA Arrival Event

The DATA exit routine is scheduled when a normal data indication is generated and no TRECVC service request is outstanding. A normal data indication is generated by the API on receiving a T-DATA.indication primitive from the transport provider. Normal data indications can only be generated on endpoints in the connected (TSCONNECT) or release-in-progress (TSOURLSE) state.

The DATA exit routine should execute a TRECVC macro instruction to receive the data buffered for the endpoint. Alternatively, the DATA exit routine can defer this responsibility to the mainline program by posting an ECB on which it is waiting.

Note: Optimum performance is achieved by receiving the data as soon as possible.

Once the DATA exit routine is scheduled, it is not rescheduled until all available data is received by the application program. If another T-DATA.indication primitive is issued by the transport provider and some previous data has not yet been received, new data is buffered at the endpoint without rescheduling the DATA exit routine. Therefore, once the DATA exit routine is entered, sufficient TRECVC service requests should be issued to receive all of the available data. A bit (TOMORE) is set in the option code field (TPLOPCD2) of the TPL to indicate more data is available to be received (see the description of the TRECVC macro instruction in the *Unicenter TCPaccess Communications Server Assembler API Macro Reference*).

Note: Expedited data can arrive at the endpoint and be received by subsequent TRECVC macro instructions. A bit (TOEXPDTE) is set in the option code field (TPLOPCD2) to indicate expedited data was received. See [XDATA Event](#) for more information on expedited data.

A CLTS endpoint operates in a similar fashion. The DATA exit routine is scheduled when a datagram indication is generated and no TRECVCFR service request is outstanding. In this case, a datagram indication is generated by the API on receiving a T-UNITDATA.indication primitive from the transport provider. Datagram indications can only be generated on CLTS endpoints that are in the disabled (TSDSABLD) state. Once the DATA exit routine is scheduled, it is not rescheduled until all buffered datagrams are received by the application program.

DGERR Event

The DGERR exit routine is scheduled whenever the transport provider indicates to the API that it could not deliver a datagram previously transferred with a TSENDTO service request. The transport provider detected the error after the TSENDTO request completed successfully.

Note: Datagram error indications can only occur on CLTS endpoints that are in the disabled (TSDSABLD) state.

The DGERR exit routine should execute a TRECVCERR macro instruction to receive the protocol address and protocol options associated with the datagram in error. A protocol-dependent datagram error code is also returned (see the *Unicenter TCPaccess Communications Server Assembler API Macro Reference* for more information).

Note: Connectionless-mode service is unreliable. The fact that the DGERR exit routine is not entered does not imply that a particular datagram was delivered successfully. Generally, errors that cause the DGERR exit routine to be entered are detected locally before the datagram is transmitted onto the network, such as an invalid protocol address.

DISCONN Event

The DISCONN exit routine is scheduled in response to receiving a T-DISCONNECT.indication primitive from the transport provider.

A disconnect indication can be generated for a COTS endpoint that is in one of the following states:

- Connect-indication-pending (TSINCONN)
- Connect-in-progress (TSOUCONN)
- Connected (TSCONNECT)
- Release-indication-pending (TSINRLSE)
- Release-in-progress (TSOURLSE)

The disconnect indication serves to notify the application program that an established connection was released or a connection attempt was abandoned. The exit routine should execute a TCLEAR macro instruction that returns the endpoint to the enabled or disabled state. Alternatively, the exit routine may post an ECB waited on by the mainline program, and the mainline program can issue the TCLEAR macro instruction. The TCLEAR macro instruction receives information associated with the disconnect indication, including disconnect user data and a protocol-dependent disconnect reason code.

Only one disconnect indication can be generated per established connection or connection attempt. A disconnect indication is never generated for a CLTS endpoint.

RELEASE Event

The RELEASE exit routine is scheduled when a release indication is generated and no TRELACK service request is outstanding. A release indication is generated by the API on receiving a T-RELEASE.indication primitive from a transport provider that supports orderly release of a connection. Release indications can only be generated on endpoints in the connected (TSCONNECT) or release-in-progress (TSOURLSE) state.

The RELEASE exit routine should execute a TRELACK macro instruction to acknowledge the release indication. Alternatively, the exit routine may post an ECB on which the mainline program is waiting, and the mainline program can acknowledge the release indication.

The RELEASE exit routine is not scheduled until all available data is received by the application program. After the release indication is received, no more TRECVC macro instructions should be executed at the endpoint. However, the endpoint can continue to send data until a TRELEASE macro instruction is executed.

Only one release indication is generated per established connection, and can only be generated on endpoints for which the orderly release option was selected. If the endpoint is operating in connectionless mode using associations, the optional orderly release service is automatically selected. In this case, a release indication is simulated when the application program executes a `TRELEASE` macro instruction.

Send Window
Opened Protocol
Event

The Send Window Opened protocol event is scheduled when an endpoint is defined with `MODE=SOCKETS` and the event is defined in the `TEVENTLST` or `TEXTLST` macro. The exit can be specified on the `TEVNTLST` or `TEXTLST` macro, where `SENDWIND=parameter`. The ECB can be specified on the `TEVNTLST` macro, where `SENDWIND=(<addr>, ECB)`.

After a series of `TSEND` or `TSENDTO` requests have used all of the available send-buffer space, the Send Window Opened protocol event is armed. The protocol event is triggered when send buffer space becomes available. TCP acknowledgment (ACK) of sent data causes buffer space to become available. In the case of UDP, buffer space is released as soon as datagrams using the buffer space are placed on the network.

XDATA Event

The XDATA exit routine is scheduled when an expedited data indication is generated and no `TRECV` service request is outstanding. An expedited data indication is generated by the API on receiving a `T-EXPEDITED-DATA.indication` primitive from the transport provider. Expedited data indications can only be generated on COTS endpoints in the connected (`TSCONNECT`) or release-in-progress (`TSOURLSE`) state.

The XDATA exit routine should execute a `TRECV` macro instruction to receive the data buffered for the endpoint. Alternatively, the XDATA exit routine can defer this responsibility to the mainline program by posting an ECB on which it is waiting. However, optimum performance is achieved by receiving the data as soon as possible.

Once the XDATA exit routine is scheduled, it is not rescheduled until all available data is received by the application program. If another `T-EXPEDITED-DATA.indication` primitive is issued by the transport provider and some previous data was not received, new data is buffered at the endpoint without rescheduling the XDATA exit routine. Therefore, once the XDATA exit routine is entered, sufficient `TRECV` service requests should be issued to receive all of the available data. A bit (`TOMORE`) is set in the option code field (`TPLOPCD2`) to indicate more data is available to be received (see the *Unicenter TCPaccess Communications Server Assembler API Macro Reference* for more information).

For scheduling exit routines, expedited data is treated as normal data if no XDATA exit routine is specified. If the DATA and XDATA exit routines have both been specified, an expedited data indication is generated if no expedited data is available to be received and new expedited data arrives. However, if expedited data is pending, newly arriving normal data does not generate a normal data indication, and the DATA exit routine is not scheduled.

Scheduling of DATA and XDATA Exit Routines

The following table summarizes when the DATA and XDATA exit routines are scheduled.

Note: The column labeled Data Pending indicates whether data of the specified type is already available when new data arrives at the endpoint. Expedited data is not supported for CLTS endpoints operating in association mode.

Exit Routine Specified		Data Pending			Exit Routine Scheduled
Data	Xdata	Normal	Expedited	New Data	Scheduled
No	No	No/Yes	No/Yes	Any	None
No	Yes	No/Yes	No/Yes	Normal	None
		No/Yes	No	Expedited	XDATA
		No/Yes	Yes		None
Yes	No	No	No	Any	DATA
		No	Yes		None
		Yes	No		None
		Yes	Yes		None
Yes	Yes	No	No	Normal	DATA
		No	Yes		None
		Yes	No		None
		Yes	Yes		None
		No	No	Expedited	XDATA
		No	Yes		None
		Yes	No		XDATA
		Yes	Yes		None

SYNAD/LERAD—Synchronous Error Recovery Exits

The API supports two synchronous error recovery exit routines:

- The SYNAD exit routine: Used to handle physical errors and exceptional conditions
- The LERAD exit routine: Used to handle program logic errors

If the appropriate error recovery exit routine is not provided by the application program, the detection and recovery from such errors must be handled inline with the macro instruction. The SYNAD and LERAD exit routines are specified in the AOPEN exit list and apply to all endpoints opened by a given transport user (that is, the application program task).

The SYNAD or LERAD exit routine can be entered at two different points during the execution of any TPL-based service request:

- If the initial request for the operation fails (after it is rejected)
- If the request is accepted, after the operation fails (after it completes abnormally)

If the TPL-based macro instruction is executed in asynchronous mode, then the first point occurs just prior to returning control to the next sequential instruction following the TPL-based macro instruction, and the second point occurs just prior to returning control to the next sequential instruction following the corresponding TCHECK macro instruction.

If the TPL-based macro instruction is executed in synchronous mode, then the TCHECK function is embedded, and the application program cannot distinguish at which point entry was made.

In either case, entry into a SYNAD or LERAD exit routine is made no more than once for any given instance of a TPL-based macro instruction.

The SYNAD or LERAD exit routine is entered with the TPL address in register one, and a recovery action code in register zero. The recovery action code can be used to determine error recovery procedures. This is the same recovery action code that is returned to the application program if a SYNAD or LERAD exit routine is not specified. The value of the recovery action code also determines which exit routine is entered. Therefore, the SYNAD and LERAD exit list addresses can specify a common exit routine, or separate exit routines tailored for their individual use.

Register 14 contains an address in the API that the exit routine should branch to if it desires to return control to the next sequential instruction in the application program. If control is returned, the contents of registers zero and 15 is passed through to the application program, and the contents of registers 2-12 is restored from the save area whose address is in register 13. The exit routine should take care not to modify the contents of the save area or the save area address. If the exit routine requires its own save area, one should be allocated, and the contents of register 13 should be restored before returning to the API.

On entry to the SYNAD or LERAD exit routine, registers 2-12 contain the application program's general-purpose registers at the time the initial request was issued or when the TCHECK macro instruction was executed, depending at which point the exit routine was called. The exit routine can take advantage of this fact if it knows how those registers are used by the application program. The exit routine is free to use the registers for any purpose without restoring them before returning to the API.

If the TCHECK or TPL-based macro instruction that caused entry into the SYNAD or LERAD exit routine used one of the general registers between two and 12 inclusive to contain the TPL address, bit zero of that register is copied into bit zero of register one. If the mainline program assures that bit zero is always reset when issuing a TPL-based request, and the exit routine assures bit zero is set when issuing such a request, recursion can be detected by testing register one. Any SYNAD or LERAD exit routine that issues TPL-based requests should test for recursion or risk entering an endless loop.

SYNAD/LERAD Errors

There are some circumstances when the SYNAD or LERAD exit routine cannot be entered. This happens when a fatal error occurs before the proper environment can be established to call the exit routine. In this case, control is always returned to the next sequential instruction in the application program.

The general return code in register 15 is greater than four, indicating that a fatal error occurred. This usually occurs before a TPL-based request is accepted, but can also occur on a TCHECK macro instruction if the TPL or some internal control block has become corrupted.

The following are examples of fatal errors that prevent scheduling of the SYNAD or LERAD exit routine:

- An invalid function code was detected
- The TPL does not contain a valid control block identifier, or resides in store-protected memory
- The TPL does not contain a valid endpoint ID
- The APCB or internal API control blocks have become corrupted
- The APCB is closed

If control is returned to the API, register 15 should contain a general return code that the application program can test for success or failure. If the exit routine was able to recover from the error, a zero should be returned to indicate success. This appears to the application program as if no error occurred. Otherwise, a non-zero value should be returned. We recommend that the exit routine use values consistent with the design of the API.

If the exit routine chooses not to return, control continues as if there were a branch to the exit routine immediately following the TCHECK or TPL-based macro instruction. However, this is permitted only if the TCHECK or TPL-based macro instruction was executed by the mainline program, or an extension thereof. If the SYNAD or LERAD exit routine was entered from a macro instruction executed by an asynchronous exit routine, be careful to always return to the API so the asynchronous exit routine can also return control.

LERAD Exit Routine

The LERAD exit routine is entered when a program logic error occurs.

Errors of this type are generally detected early before a TPL-based request is accepted by the API. The recovery action code in register zero is set to one of the following values:

Name	Dec	Hex	Type Of Error
TAFORMAT	16	X'10'	Format or specification error.
TAPROCED	20	X'14'	Sequence or procedural error.
TATPLERR	24	X'20'	Logic error with no TPL return code.

TAFORMAT and TAPROCED errors enter the LERAD exit routine with the TPL set inactive and the TPLRTNCD field containing valid return code information. The specific error code (TPLERRCD) and diagnostic code (TPLDGNCD) contain additional information regarding the error.

If the recovery action code is set to TATPLERR, the TPL is still active and return-code information could not be stored. This recovery action code can occur as the result of one of the following circumstances:

- A TPL-based request was issued and the associated TPL was active
- A TCHECK macro instruction was executed before the TPL was marked complete.

The second circumstance can only occur for asynchronous requests for which a TPL exit routine was specified. That is, the TCHECK macro instruction was executed before the exit routine was entered. For this recovery action code there is no way for the exit routine to differentiate the cause of the error.

Program logic errors typically occur while an application program is being developed and debugged. Hopefully, once debugging is complete, errors of this type do not occur. Therefore, it is unusual for a LERAD exit routine to attempt to recover from such errors. The best course of action is to dump the address space and terminate the application program.

SYNAD Exit Routine

The SYNAD exit routine is entered when a physical error or exceptional condition occurs. Unlike program logic errors, which tend not to occur after a program is debugged, these errors can happen at any time and often occur after a TPL-based request is accepted by the API.

The recovery action code in register zero is set to one of these values:

Name	Dec	Hex	Type Of Error
TAEXCPTN	4	X'04'	Exceptional condition
TAINTEG	8	X'08'	Connection or data integrity error
TAENVIRO	12	X'0C'	Environmental error

The SYNAD exit routine is always entered with the TPL set inactive and the TPLRTNCD field containing valid return-code information stored by the API. A copy of the recovery action code is stored in the TPL (TPLACTCD) along with a specific error code (TPLERRCD) and a diagnostic code (TPLDGNCD). This information can be used to determine more precisely the particular cause of the error.

The SYNAD exit routine may choose to merely log the occurrence of an error and record pertinent diagnostic information, or may attempt to recover from the error. If necessary, additional TPL-based macro instructions may be issued to affect recovery. However, the exit routine should implement an appropriate mechanism (see [SYNAD/LERAD – Synchronous Error Recovery Exits](#)) to detect recursion caused by reentry into the exit routine. If recovery from the error is successful, registers 15 and zero can be set accordingly so that the mainline program is unaware of the occurrence of the error, and processing can continue as usual.

TPEND Exit Routine or ECB

The TPEND exit routine is entered when the transport provider terminates (or is about to terminate) and can no longer provide service to the application program. The TPEND exit routine is entered asynchronously with register one containing the address of a TXP.

The TXP identifies the exit type (TXPTYPE) as a TPEND exit (TXPTPEND), the endpoint ID (TXPEPID) is set to zero since no endpoint is associated with this event, and the exit routine parameter (TXPPARM) gives the reason for the termination. The reason code is in multiples of four and can be used as a branch or table index to locate the appropriate processing routine.

These reason codes are defined as follows.

Name	Dec	Hex	Reason For Termination
TXPRDRAN	0	X'00'	Operator drained subsystem.
TXPRSTOP	4	X'04'	Operator stopped subsystem.
TXPRTERM	8	X'08'	Subsystem abnormally terminated.

Entering the TPEND Exit Routine

The TPEND exit routine is normally entered three times.

- The first time occurs when the system operator enters a command from the operator's console to drain the API subsystem.

This command is entered in anticipation of shutting down the subsystem. Existing transport service users are allowed to continue normal operation, but new transport users are not allowed to establish sessions with the transport provider. That is, AOPEN macro instructions are completed with an error (APCBEDRA).

When the TPEND exit routine is entered with the reason code set to TXPRDRAN, a variable should be set to prevent the application program from attempting to define any new transport service users. That is, prevent AOPEN macro instructions from being issued. If the application program is serving interactive users, they should be informed by whatever means are appropriate that the network subsystem may be shut down soon, and should be encouraged to complete their use of any network resources accessed via the API.

This entry into the TPEND exit routine only serves as a warning that a shutdown of the subsystem is about to happen. Any TPL-based request issued after the TPEND exit routine is entered is completed conditionally. That is, if no other errors occur and the function is allowed in drain mode, the request is completed normally with TRSTOP set in the conditional completion code returned in the TPL (and register zero). Any request that is not allowed in drain mode is completed with TRFAILED/TEDRAIN.

- The second entry into the TPEND exit routine occurs after the TPEND exit is entered in drain mode for all endpoints.

In this case, the shutdown process has actually begun, and the application program has a limited time to complete its use of the API services. No new endpoints can be opened, but existing endpoints are allowed to continue issuing certain TPL-based service requests required to shutdown the application. Any allowable TPL-based request issued after the TPEND exit routine is entered is completed conditionally. That is, if no other errors occur, the request is completed normally with TCSTOP set in the conditional completion code returned in the TPL (and register zero). Any request that is not allowed during the stop phase is completed with TRFAILED/TESTOP.

- The third and last entry into the TPEND exit routine occurs when the API subsystem actually terminates, either normally or abnormally.

A second stop command can be issued to force this phase of termination. At this point, the application program is only allowed to clean up resources in its address space by issuing TCLOSE and ACLOSE macro instructions. In the case of a graceful shutdown, the application program should have already closed its endpoints and terminated its session with the API by the time subsystem termination occurs.

Note: The TPEND exit is not guaranteed to be entered all three times.

Example

If the system operator stops the API subsystem without first issuing a drain command, the first entry does not occur. Similarly, if the subsystem is canceled or abnormally terminates without a stop command, the second entry does not occur.

If the application program does not specify a TPEND exit routine in its AOPEN exit list, no asynchronous notification is given for the events previously described. The application program has to rely on return-code information stored in the TPL to detect the occurrence of a shutdown or subsystem termination.

The TPEND exit routine for the API differs from the VTAM TPEND exit in that all instances of the TPEND exit routine are scheduled by the API at the same priority. Whereas VTAM schedules the termination instance at a higher priority (when the reason code is X'08', VTAM schedules the TPEND exit routine so that it preempts other asynchronous exit routines). For the API, all asynchronous exit routines are executed serially, including all instances of the TPEND exit routine.

APEND Exit Routine

The APEND exit routine is entered when the application subsystem is shut down and can no longer provide service to the application program. The APEND exit routine is entered asynchronously with register one containing the address of a TXP. The TXP identifies the exit type (TXPTYPE) as an APEND exit (TXPAPEND). Instead of an endpoint ID TXPAPCB is set to the APCB associated with this event, and the exit routine parameter (TXPPARM) gives the reason for the termination. The reason code is in multiples of four and can be used as a branch or table index to locate the appropriate processing routine.

The following reason codes are defined:

Name	Dec	Hex	Reason For Termination
TXPRDRAN	0	X'00'	Operator drained subsystem.
TXPRSTOP	4	X'04'	Operator stopped subsystem.
TXPRTERM	8	X'08'	Subsystem abnormally terminated.

Entering the APEND Exit Routine

The APEND exit routine is normally entered three times.

- The first time occurs when the system operator enters a command from the operator's console to drain the API subsystem.

This command is entered in anticipation of shutting down the subsystem. Existing transport service users are allowed to continue normal operation, but new transport users are not allowed to establish sessions with the transport provider. That is, AOPEN macro instructions are completed with an error (APCBEDRA).

When the APEND exit routine is entered with the reason code set to TXPRDRAN, a variable should be set to prevent the application program from attempting to define any new transport service users. That is, prevent AOPEN macro instructions from being issued. If the application program is serving interactive users, they should be informed by whatever means are appropriate that the network subsystem may be shut down soon, and should be encouraged to complete their use of any network resources accessed via the API. This entry into the APEND exit routine only serves as a warning that a shutdown of the subsystem is about to happen. Any TPL-based request issued after the APEND exit routine is entered, are completed conditionally. That is if no other errors occur and the function is allowed in drain mode, the request is completed normally with TRSTOP set in the conditional completion code returned in the TPL (and register zero). Any request that is not allowed in drain mode is completed with TRFAILED/TEDRAIN.

- The second entry into the APEND exit routine occurs after the APEND exit is entered in drain mode for all endpoints. In this case, the shutdown process has actually begun, and the application program has a limited time to complete its use of the API services. No new endpoints can be opened, but existing endpoints are allowed to continue issuing certain TPL-based service requests that are required to shutdown the application. Any allowable TPL-based request issued after the APEND exit routine is entered is completed conditionally. That is, if no other errors occur, the request is completed normally with TCSTOP set in the conditional completion code returned in the TPL (and register zero). Any request that is not allowed during the stop phase is completed with TRFAILED/TESTOP.
- The third and last entry into the APEND exit routine occurs when the API subsystem actually terminates, either normally or abnormally. A second stop command can be issued to force this phase of termination. At this point, the application program is only allowed to clean up resources in its address space by issuing TCLOSE and ACLOSE macro instructions. In the case of a graceful shutdown, the application program should have already closed its endpoints and terminated its session with the API by the time subsystem termination occurs.

Note: The APEND exit is not guaranteed to be entered all three times.

Example

If the system operator stops the API subsystem without first issuing a drain command, the first entry does not occur. Similarly, if the subsystem is canceled or abnormally terminates without a stop command, the second entry does not occur.

If the application program does not specify an APEND exit routine in its AOPEN exit list, no asynchronous notification is given for the events previously described. The application program must rely on return-code information stored in the TPL to detect the occurrence of a shutdown or subsystem termination.

The APEND exit routine for the API differs from the VTAM TPEND exit in that all instances of the APEND exit routine are scheduled by the API at the same priority. Whereas VTAM schedules the termination instance at a higher priority (when the reason code is X'08', VTAM schedules the TPEND exit routine so that it preempts other asynchronous exit routines). For the API, all asynchronous exit routines are executed serially, including all instances of the APEND exit routine.

Deriving Context in Exit Routines

Generally, the information passed to an exit routine is sufficient for providing the context the application program needs for processing the event.

Example 1 The TPL address passed to SYNAD, LERAD, and TPL exit routines identifies the particular request causing entry into the exit routine.

Similarly, the endpoint ID provided in the TXP for protocol events identifies the endpoint at which the event occurred. However, when exit routines are shared between multiple tasks, additional context may be required.

Asynchronous exit routines that are provided the address of a TXP in register one can acquire additional context from the TXP itself.

Example 2 TXPACNTX is a word of context related to the transport user. It is acquired from the APCB and is specified by the ACNTX parameter. It can be an arbitrary fullword value, and is not interpreted by the API.

Similarly, TXPUCNTX is related to the endpoint and is specified when the endpoint is opened by coding the UCNTX parameter on the TOPEN macro instruction. The API merely copies this information into the TXP before the exit routine is entered.

The TXPECNTX field provides context for the language environment. If a higher-level language environment is not being used (ENVIRO=ASM is coded on the APCB macro instruction), this field can be used by the application program by specifying the ECNTX parameter on the APCB macro instruction.

For those exit routines that are not provided a TXP, a different technique can be used. In this case, the exit routine is always provided the address of a TPL. Since the application program provides the TPL, it can arrange to have the TPL located relative to other information.

Example 3 The address of the TPL can be decremented by a known amount to obtain the base of an application program control block. That is, the TPL is located at some fixed offset within the control block, or the application program can place information at the end of the TPL (that is, the TPL is extended in length to include application program information).

Handling Errors and Special Conditions

This section describes two types of TPL-based requests:

- Synchronous

For synchronous TPL-based requests, a single macro instruction is issued. On return to the application program, error or exceptional-condition information about the requested operation is available.

- Asynchronous

For asynchronous requests, two TPL-based macro instructions are required:

- A request macro instruction
- A TCHECK macro instruction

Error and exceptional-condition information can thus be returned at two different stages:

- As a result of the request for the operation being accepted
- If the request is accepted, as a result of the operation completing successfully or unsuccessfully

Macro Information

Following a TPL-based macro instruction, information is available to the application program about the acceptability of the request or about the completion of the operation. This information can be provided by the API; or, if an error or exceptional condition was detected and the API invoked the program's SYNAD or LERAD exit routine, register information is provided by the exit routine.

This information consists of:

- A return code in register 15 (termed the general return code)
- In some cases a return code in register zero (termed either a recovery action code or a conditional completion code)
- Information in the TPL

The information stored in the TPL consists of a copy of the recovery action code and conditional completion code, and when an error occurs, it also consists of a specific error code and a diagnostic code.

If an error or exceptional condition occurs, it can be analyzed and handled in either the mainline program or exit routine in which the TPL-based request was issued, or in the SYNAD or LERAD exit routine designated by the AOPEN exit list. In either case, the analysis is performed by examining the return code information provided in registers and stored in the TPL. These return codes are discussed in more detail in [General Return Codes](#) .

General Return Codes

The following table provides a list of the general return codes

Name	Dec	Hex	Meaning
TROKAY	0	X'00'	Request accepted or completed successfully.
TRFAILED	4	X'04'	Request not accepted or completed abnormally.
TRFATLFC	8	X'08'	Fatal error: invalid function code.
TRFATLPL	12	X'0C'	Fatal error: invalid TPL.
TRFATLAM	16	X'10'	Fatal error: internal access method error.
TRFATLAP	20	X'14'	Fatal error: APCB is closed or invalid.

The general return code also indicates what other information is available in register zero and the return code field of the TPL.

Summary of Register and TPL Return Codes

The following table summarizes the relationship of the general return code in register 15 and other information returned to the application program.

TPL Return Codes				SYNAD or LERAD Exit Routine Entered
Register 15	Register 0	TPLACTCD	TPLERRCD	
TROKAY X'00'	Conditional completion code	Recovery action code (X'00')	Conditional completion code	No
TRFAILED X'04'	Recovery action code or value from exit routine	Recovery action code (X'04' - X'14')	Specific error code	Yes
TRFATLFC X'08'	TPL function code	No information stored	No information stored	No
TRFATLPL X'0C'	Diagnostic code	No information stored	No information stored	No

TPL Return Codes				SYNAD or LERAD Exit Routine Entered
Register 15	Register 0	TPLACTCD	TPLERRCD	
TRFATLAM X'10'	Diagnostic code	No information stored	No information stored	No
TRFATLAP X'14'	Diagnostic code	No information stored	No information stored	No

Conditional Completion Codes

When a TPL-based request completes successfully (the general return code is zero), a conditional completion code may be returned in register zero.

- If the value returned in register zero is zero, then the request is said to have completed normally
- If the value returned in register zero is non-zero, then the request is said to have completed conditionally, and the value in register zero is the conditional completion code

The purpose of the conditional completion code is to notify the application program of the occurrence of some condition that although it did not prevent the successful completion of the requested operation, may need to be acted on by the application program. More than one condition can be present at one time, and as such, each bit in the conditional completion code represents a different condition.

The following conditions are recognized by the API.

Name	Dec	Hex	Meaning
TCOKAY	0	X'00'	Normal (unconditional) completion.
TCVERIFY	128	X'80'	One or more protocol options did not verify.
TCNEGOT	64	X'40'	Protocol options negotiated to inferior value.
TCTRUNC	32	X'20'	Data truncated to fit in storage area.
TCSTOP	8	X'08'	Subsystem shutdown in progress.

When a TPL-based request completes successfully, the recovery action code stored in the TPL (TPLACTCD) is set to zero (X'00'), and the conditional completion code is stored in the specific error code field (TPLERRCD). Assertion of a conditional completion code does not cause the SYNAD or LERAD exit routine to be entered.

Recovery Action Codes

Recovery action codes serve the following purposes:

- They are used by the API to determine whether the SYNAD or LERAD exit routine should be entered following the unsuccessful completion of a request
- They provide the exit routine a way to distinguish a class of errors and to dispatch the appropriate processing routine.

The following recovery actions codes are used by the API.

Name	Dec	Hex	Meaning
TAOKAY	0	X'00'	Request completed successfully.
TAEXCPTN	4	X'04'	Failed due to exceptional condition.
TAINTEG	8	X'08'	Connection or data integrity error.
TAENVIRO	12	X'0C'	Environmental error.
TAFORMAT	16	X'10'	Format or specification error.
TAPROCED	20	X'14'	Sequence or procedural error.
TATPLERR	24	X'18'	Logic error with no TPL return code.

Recovery Action Code Classification for Errors

Errors are classified by recovery action code as follows.

Errors with the following recovery action codes are classified as physical errors or exceptional conditions that cause the SYNAD exit routing to be entered (if one is specified):

- TAXCPTN
- TAINTEG
- TAENVIRO

Errors with the following recovery action codes are classified as program logic errors that cause the LERAD exit routine to be entered:

- TAFORMAT
- TAPROCED
- TATPLERR

TAOKAY is the recovery action code used for successful completion.

Normally, the recovery action code is stored in the TPLACTCD field of the TPL in addition to being returned in register zero. However, the TATPLERR recovery action code is a special case. For all errors of this class, the TPL is busy with another request and cannot be modified. Therefore, the recovery action code and the accompanying specific error and diagnostic codes are not stored in the TPL.

Specific Error Codes

The specific error code is used to indicate a particular error within the class of errors defined by the recovery action code. The specific error code is only returned in the TPLERRCD field of the TPL. Since return codes are not stored for the TATPLERR recovery action code, specific error codes in this class can never be returned to the application program.

Specific error codes have been carefully selected to have generic applicability across a variety of transport providers.

Example

TEPROTO is the symbolic name of the specific error code used to indicate a protocol error. This code does not indicate a specific protocol error, but only that a protocol error occurred. Therefore, application programs that make use of the specific error code can expect a degree of commonality from one transport provider to another.

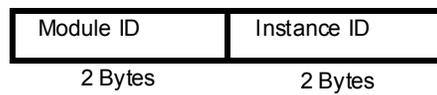
Specific error codes used by the API are documented in *Unicenter TCPaccess Communications Server Unprefixed Messages and Codes*.

Diagnostic Codes

The diagnostic code is used to indicate a particular instance of a specific error code.

Diagnostic codes are provider-dependent, and an application program that is intended to be portable between transport providers should not analyze this code. However, it is a good idea to include the diagnostic code with any information recorded for diagnostic purposes.

The diagnostic code is four bytes in length. The first two bytes are the module ID, followed by a two-byte serial number that identifies a particular error within the module:



The four-byte diagnostic code appears in API traces and is recorded in the extended diagnostic area if TPL extensions are used.

When returned to the standard TPL, the diagnostic code is uniquely mapped to a two-byte code.

The first byte identifies the module that encountered the error. The second byte identifies a specific error instance identifier within the module.

Diagnostic codes that can accompany specific error codes are listed in *Unicenter TCPAccess Communications Server Unprefixed Messages and Codes*.

AOPEN and ACLOSE Errors

After issuing the AOPEN or ACLOSE macro instruction, register 15 should be tested.

- If the return code in register 15 is zero, the APCB was opened or closed as requested.
- If the return code in register 15 is not equal to zero, the APCB was not properly opened or closed. When this occurs, register zero contains an error code.

This error code can also be returned in the APCBERRC field of the APCB. Whether or not the error code is stored depends on the value returned in register 15. If the error code is stored, it is accompanied by a diagnostic code stored in the APCBDGNC field.

The API uses the following AOPEN and ACLOSE return codes.

Dec	Hex	Meaning
0	X'00'	The request operation was successful.
4	X'04'	No operation was performed (no error code stored).
8	X'08'	A temporary failure occurred.
12	X'0C'	A permanent failure occurred.
16	X'10'	A fatal error occurred (no error code stored).

AOPEN and ACLOSE errors must be handled in the mainline program (that is, the SYNAD or LERAD exit routine is not entered when such errors occur).

- If the failure was temporary, the request can be retried after some delay
- If the failure was permanent, the request should not be retried unless the permanent error flag set in the APCB is cleared before reissuing the request

AOPEN and ACLOSE error codes are documented in *Unicenter TCPaccess Communications Server Unprefixed Messages and Codes*.

Application Program Organization

The API provides many facilities for program control and synchronization. Most of these facilities are optional and are not required to be used by every application program. The facilities that are used by a particular program depends in large part on the environment in which it operates and how it is organized.

Types of Organization

An application program is generally organized along one of these lines:

- Using TPL completion ECBs as the primary mechanism for synchronization and control.

All TPL-based requests should be executed from the mainline program and little use (if any) should be made of exit routines except for error recovery. An application program organized in this manner issues a service request in anticipation of some event, and then waits for it to complete.

This is useful when a TREC macro instruction is executed without knowing whether data was available, and at some point, the program must wait for completion.

- Organizing the application program to respond to events instead of anticipating their occurrence.

An application program organized in this manner primarily uses asynchronous exit routines or protocol event ECBs.

This happens when a DATA exit routine is defined, and when data arrives the exit routine issues the TRECVC macro instruction.

An application program can also combine both approaches. However, when using ECB posting with asynchronous exits, the program must be prepared to handle certain anomalies that may occur. In particular, the time sequence of events may appear out of order.

Example

Assume a DATA exit was specified for an endpoint operating in connection mode. Further, assume that a TCONNECT macro instruction completed successfully, and a TCONFIRM macro instruction was issued that specifies ECB posting. When the connect confirmation arrives, the TCONFIRM request completes and the ECB is posted. However, if data arrives at the endpoint before the application program's address space is dispatched, the DATA exit routine may be entered before the program detects that the TCONFIRM request completed. The application program must be able to handle this situation, or should be coded to avoid it.

Multitasking Operation Rules

The API is designed to support both multitasking operation and multiple address space operation.

These rules apply to using the API in a multitasking environment:

- The task that issues the ACLOSE macro instruction must be the task that opened the APCB.
- All asynchronous exit routines are entered from an IRB on the TCB of the task that opened the APCB to which the exit list is linked.
- All asynchronous exit routines are entered from an IRB on the TCB of the task that opened the endpoint to which the exit list is linked.
- All asynchronous exit routines associated with a given task are executed at the same priority. That is, asynchronous exit routines are serialized at the task level.
- If desired, other tasks can issue TPL-based requests (other than TOPEN and TCLOSE) for any given endpoint. However, exits continue to execute on the owning task, as mentioned in the previous two rules.

- Exits associated with an endpoint that is passed with a TCLOSE OPTCD=PASS macro are driven in the task that issues the corresponding TOPEN OPTCD=OLD when the TOPEN completes.
- A task that relinquishes control of an endpoint by issuing a TCLOSE macro instruction (with OPTCD=PASS) is treated as if it had never opened the endpoint at all.
- The task that receives control of an endpoint by issuing a TOPEN macro instruction (with OPTCD=OLD) becomes the owning task, and is treated as if it originally opened the endpoint.

Multiple Address Spaces

The API supports operation across multiple address spaces. In an address space, the rules for multitasking operation, described in the *Unicenter TCPaccess Communications Server Assembler API Macro Reference* apply.

An additional facility is provided to pass sessions from one address space to another. Using this facility involves establishing a session up to some state (with no active data transfer in progress) and issuing a TCLOSE macro with OPTCD=PASS. Optionally, the user can specify which address space, by ASCB address, can receive the session. The receiving program issues a TOPEN macro specifying the original endpoint ID, the OPTCD=OLD, the ASCB address of the address space that is passing the endpoint, and optionally the TCB address of the task that is passing the endpoint. The endpoint does not need to be in common storage.

Do not assume that the endpoint ID remains the same after the TOPEN OPTCD=OLD is issued. As in any TOPEN invocation, the endpoint ID token should be copied on completion of the TOPEN and used in all subsequent requests that refer to the endpoint.

The application can also specify ASCB=ANY on the TCLOSE OPTCD=PASS macro to indicate that the endpoint is eligible for passing to any address space that issues a matching TOPEN OPTCD=OLD request..

Initialization of the second address space, passing the required information, and synchronization of the two address spaces is up to the user.

24-Bit and 31-Bit Addressing

The API supports operation in either 24-bit addressing mode or 31-bit addressing mode. The addressing mode in effect at the time the AOPEN macro is issued determines the addressing mode for all further operations.

Exits also run in this addressing mode. Interface routines are placed in 24-bit memory in order to accept calls in either mode. Control blocks are allocated in the addressing mode in effect when the AOPEN macro is issued.

Endpoint State Transitions

This appendix lists and describes all of the state transitions that may occur at an endpoint.

The following topics are discussed:

- [Defined Endpoint States](#) – Briefly describes the defined endpoint states
- [The State Transition Tables](#) – Includes array tables that show how endpoint and current state are used to determine new endpoint states

Defined Endpoint States

A machine-readable endpoint state transition table is interpreted by API routines to manage a state variable maintained for each endpoint. This state variable is used to control the execution sequence of API service functions. The state variable can be read with the TSTATE function, and is returned as part of a state word defined by the TSW DSECT.

These nine endpoint states are defined:

TSCLOSED	Closed.
TSOPENED	Opened.
TSDSABLD	Disabled.
TSEENABLED	Enabled.
TSINCONN	Connect indication pending.
TSOUCONN	Connect in progress (awaiting confirm indication).
TSCONNECT	Connected (or associated).
TSINRLSE	Release indication pending.
TSOURLSE	Release in progress (awaiting release indication).

The State Transition Tables

The tables in this section list the nine state transitions that may occur at an endpoint. Each state transition table is organized as an array where each row represents an event and each column represents one of the possible endpoint states. Given some event and the current state of the endpoint, the array element at the intersection of the row and column is the new state of the endpoint. If the array element is null (blank), the event is invalid for the current state.

Each table includes the same event state information in the Event columns, and three of the nine endpoint states in the Current State columns.

Each event listed in the state table consists of the successful completion of a particular API service function, and various conditions that were in effect during its execution.

- Conditions listed in uppercase represent parameters or option codes provided with the service request.
- Conditions listed in lowercase apply to internal variables maintained by the API.
- Each event also lists the service type that must be in effect for the endpoint.

These are the variables:

qlstn The negotiated size of the connect indication queue specified by the QLSTN parameter in a successful TBIND request.

count The number of pending connect indications.

Note: These notes correspond to the numbers (for example, 3) in the tables:

1. Connectionless Transport Service (CLTS) service mode is valid only when using associations. TYPE=(CLTS,ASSOC) must have been specified when the endpoint was opened.
2. Connection-Oriented Transport Service (COTS) service mode is valid only when orderly release is supported by the transport provider. TYPE=(COTS,ORDREL) must have been specified when the endpoint was opened.
3. Control of the endpoint is passed to another task or address space. On completion of TOPEN, the state of the new endpoint is the same as the state of the old endpoint before it was closed.
4. This event represents the state transition for the endpoint to which a connection is accepted when the endpoint is different from the one receiving the connect indication.

5. The number of pending connect indications is incremented by one.
6. The number of pending connect indications is decremented by one.

Endpoint States for TSCLOSED, TSOPENED, and TSDSABLD

The following table lists the endpoint states for the TSCLOSED, TSOPENED, and TSDSABLD states.

Event	Function	Conditions	Service Type	Current State:		
				TSCLOSED	TSOPENED	TSDSABLD
TACCEPT		count=1, NEWEP=EP	COTS, CLTS,			
		count=1, NEWEP=EP	COTS, CLTS,			
		count>1, NEWEP=EP	COTS, CLTS,			
			COTS, CLTS,			TSCONNECT
TADDR		OPTCD=LOCAL	COTS, CLTS			TSDSABLD
		OPTCD=REMOTE	COTS, CLTS			
TBIND		QLSTN=0	COTS, CLTS		TSDSABLD	
		QLSTN>0	COTS, CLTS		TSEENABLED	TSEENABLED
TCLEAR		qlstn=0	COTS, CLTS,			
		qlstn>0, count=0	COTS, CLTS,			
		qlstn>0, count=1	COTS, CLTS,			
		qlstn>0, count>1	COTS, CLTS,			
TCLOSE		OPTCD=DELETE	COTS, CLTS		TSCLOSED	TSCLOSED
		OPTCD=PASS	COTS, CLTS		3	3
TCONFIRM			COTS, CLTS,			
TCONNECT			COTS, CLTS,			TSOUCONN
TDISCONN		qlstn=0	COTS, CLTS,			
		qlstn>0	COTS, CLTS,			
TINFO			COTS, CLTS		TSOPENED	TSDSABLD
TLISTEN		count<qlstn	COTS, CLTS,			
TOPEN		OPTCD=NEW	COTS, CLTS	TSOPENED		
		OPTCD=OLD	COTS, CLTS	3		

Event			Current State:		
Function	Conditions	Service Type	TSCLOSED	TSOPENED	TSDSABLD
TOPTION		COTS, CLTS		TSOPENED	TSDSABLD
TRECV		COTS, CLTS,			
TRECVERR		CLTS			TSDSABLD
TRECVFR		CLTS			TSDSABLD
TREJECT	count=1	COTS, CLTS,			
	count>1	COTS, CLTS,			
TRELACK	qlstn=0	COTS, 2			
	qlstn>0	COTS, 2			
TRELEASE	qlstn=0	COTS, 2			
	qlstn>0	COTS, 2			
TRETRACT		COTS, CLTS,			
TSEND		COTS, CLTS,			
TSENDTO		CLTS			TSDSABLD
TUNBIND		COTS, CLTS			TSOPENED
TUSER		COTS, CLTS		TSOPENED	TSDSABLD

Endpoint States for TSENABLD, TSINCONN, and TSOUCONN

The following table lists the endpoint states for the TSENABLD, TSINCONN, and TSOUCONN states.

Event			Current State:		
Function	Conditions	Service Type	TSCLOSED	TSOPENED	TSDSABLD
TACCEPT	count=1, NEWEP=EP	COTS, CLTS,		TSCONNECT, 6	
	count=1, NEWEP=EP	COTS, CLTS,		TSENABLD, 6	
	count>1, NEWEP=EP	COTS, CLTS,		TSINCONN, 6	
	4	COTS, CLTS,			
TADDR	OPTCD=LOCAL	COTS, CLTS	TSENABLD	TSINCONN	TSOUCONN
	OPTCD=REMOTE	COTS, CLTS			
TBIND	QLSTN=0	COTS, CLTS			
	QLSTN>0	COTS, CLTS			
TCLEAR	qlstn=0	COTS, CLTS,			TSDSABLD
	qlstn>0, count=0	COTS, CLTS,			
	qlstn>0, count=1	COTS, CLTS,		TSENABLD, 6	
	qlstn>0, count>1	COTS, CLTS,		TSINCONN, 6	
TCLOSE	OPTCD=DELETE	COTS, CLTS	TSCLOSED	TSCLOSED	TSCLOSED
	OPTCD=PASS	COTS, CLTS	3		
TCONFIRM		COTS, CLTS,			TSCONNECT
TCONNECT		COTS, CLTS,			
TDISCONN	qlstn=0	COTS, CLTS,			TSDSABLD
	qlstn>0	COTS, CLTS,			
TINFO		COTS, CLTS	TSENABLD	TSINCON	TSOUCONN
TLISTEN	count<qlstn	COTS, CLTS,	TSINCONN,	TSINCONN, 5	
TOPEN	OPTCD=NEW	COTS, CLTS			
	OPTCD=OLD	COTS, CLTS			
TOPTION		COTS, CLTS	TSENABLD		
TRECV		COTS, CLTS,			

Event			Current State:		
Function	Conditions	Service Type	TSCLOSED	TSOPENED	TSDSABLD
TRECVERR		CLTS			
TRECVFR		CLTS			
TREJECT	count=1	COTS, CLTS,		TSENABLD, 6	
	count>1	COTS, CLTS,		TSINCONN, 6	
TRELACK	qlstn=0	COTS, 2			
	qlstn>0	COTS, 2			
TRELEASE	qlstn=0	COTS, 2			
	qlstn>0	COTS, 2			
TRETRACT		COTS, CLTS,	TSENABLD	TSINCONN	
TSEND		COTS, CLTS,			
TSENDTO		CLTS			
TUNBIND		COTS, CLTS	TSOPENED		
TUSER		COTS, CLTS	TSENABLD		

Endpoint States for TSCONNECT, TSINRLSE, and TSOURLSE

This table lists the endpoint states for the TSCONNECT, TSINRLSE, and TSOURLSE states.

Event Function	Conditions	Current State:			
		Service Type	TSCONNECT	TSINRLSE	TSOURLSE
TACCEPT	count=1, NEWEP=EP	COTS, CLTS,			
	count=1, NEWEP=EP	COTS, CLTS,			
	count>1, NEWEP=EP	COTS, CLTS,			
	4	COTS, CLTS,			
TADDR	OPTCD=LOCAL	COTS, CLTS	TSCONNECT	TSINRLSE	TSOURLSE
	OPTCD=REMOTE	COTS, CLTS	TSCONNECT	TSINRLSE	TSOURLSE
TBIND	QLSTN=0	COTS, CLTS			
	QLSTN>0	COTS, CLTS			
TCLEAR	qlstn=0	COTS, CLTS,	TSDSABLD	TSDSABLD	TSDSABLD
	qlstn>0, count=0	COTS, CLTS,	TSENABLD	TSENABLD	TSENABLD
	qlstn>0, count=1	COTS, CLTS,			
	qlstn>0, count>1	COTS, CLTS,			
TCLOSE	OPTCD=DELETE	COTS, CLTS	TSCLOSED	TSCLOSED	TSCLOSED
	OPTCD=PASS	COTS, CLTS	3		
TCONFIRM		COTS, CLTS,			

Event	Function	Conditions	Current State:		
			Service Type	TSCONNECT	TSINRLSE
TCONNECT			COTS, CLTS,		
TDISCONN		qlstn=0	COTS, CLTS,	TSDSABLD	TSDSABLD
		qlstn>0	COTS, CLTS,	TSENABLD	TSENABLD
TINFO			COTS, CLTS	TSCONNECT	TSOURLSE
TLISTEN		count<qlstn	COTS, CLTS,		
TOPEN		OPTCD=NEW	COTS, CLTS		
		OPTCD=OLD	COTS, CLTS		
TOPTION			COTS, CLTS	TSCONNECT	
TRECV			COTS, CLTS,	TSCONNECT	TSOURLSE
TRECVERR			CLTS		
TRECVFR			CLTS		
TREJECT		count=1	COTS, CLTS,		
		count>1	COTS, CLTS,		
TRELACK		qlstn=0	COTS, 2	TSINRLSE	TSDSABLD
		qlstn>0	COTS, 2	TSINRLSE	TSENABLD
TRELEASE		qlstn=0	COTS, 2	TSOURLSE	TSDSABLD
		qlstn>0	COTS, 2	TSOURLSE	TSENABLD
TRETRACT			COTS, CLTS,		
TSEND			COTS, CLTS,	TSCONNECT	TSINRLSE
TSENDTO			CLTS		

Event		Current State:			
Function	Conditions	Service Type	TSCONNCT	TSINRLSE	TSOURLSE
TUNBIND		COTS, CLTS			
TUSER		COTS, CLTS	TSCONNCT		

Time-Sequence Diagrams

This appendix contains sequence diagrams showing the relationship between the API service requests issued at an endpoint and service primitives issued to or by the transport provider.

It covers [Diagrams](#) including the time-sequence diagrams.

Each time-sequence diagram shows a local transport user (Local TU) issuing requests and responding to events at the local transport interface (API), and a transport provider (TP) receiving request and response primitives, and issuing indication and confirm primitives.

Diagram Labeling

The transport provider is shown as a single entity, although, in actuality, there is a local and remote entity between which the protocol exchanges take place. Note that the transport interface between the remote transport provider and the peer transport user (Remote TU) is not shown.

The vertical lines delineating the transport provider represent the Transport Service Access Points (TSAPs) for the local and remote transport user. The vertical lines delineating the transport interface represent the endpoint from the perspective of the transport user and transport provider.

All interactions between the local transport user and the API are shown in terms of the service functions executed, and their normal or abnormal completions:

- The invocation of a function is labeled a request
- Its successful completion is simply labeled a completion
- An abnormal completion is indicated by error
- An asynchronous event that causes an exit routine to be scheduled is labeled an indication

Synchronous and Asynchronous Modes

Some sequences are shown in synchronous and asynchronous mode:

- Synchronous mode applies when service requests are issued synchronous with normal application program processing. Generally, the application program is running under control of a PRB.
- Asynchronous mode applies when service requests are issued asynchronous with normal processing. This mode requires use of exit routines, and requests are often issued under control of the IRB that runs the exit routine.

In synchronous mode, the time relationship between the occurrence of an event (for example, the arrival of some data) and invocation of the corresponding service function (for example, TRECVC) is unimportant. However, in asynchronous mode, the service function is generally issued in response to the event.

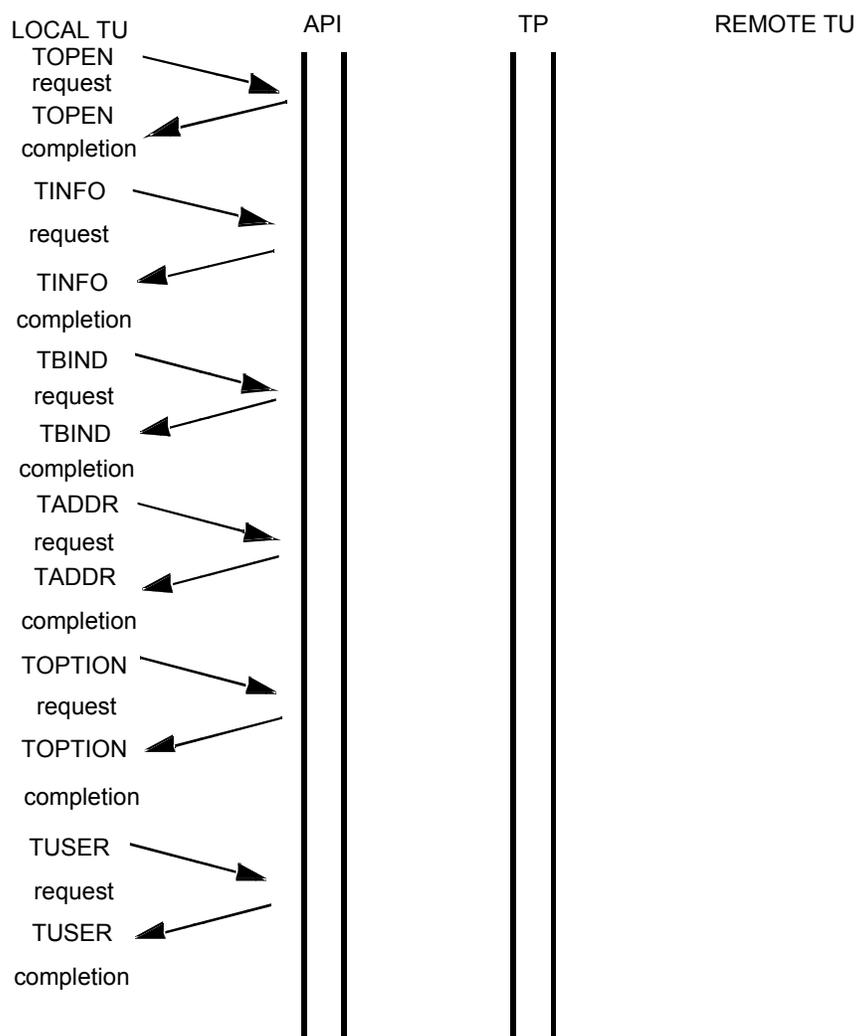
Completion and Error Events

Completion and error events occur when a TCHECK control function is executed. The TCHECK function can be executed by the API (OPTCD=SYNC) or the transport user (OPTCD=ASYN).

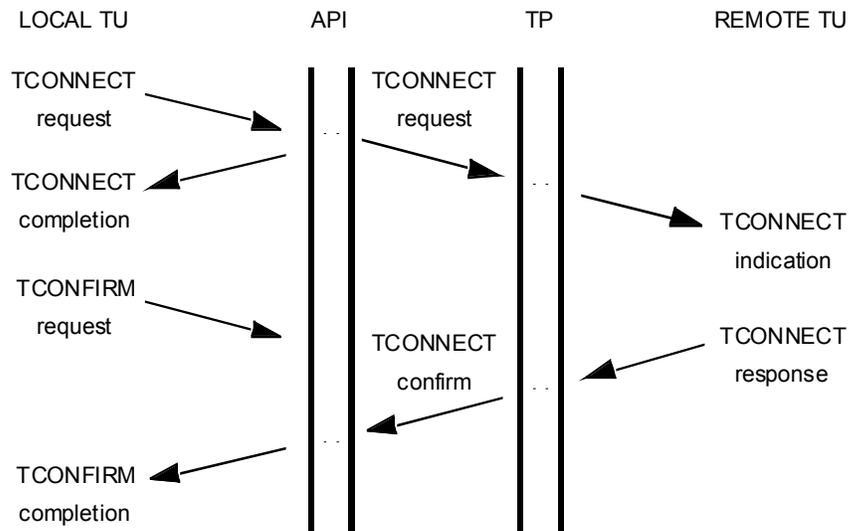
Diagrams

This section contains the time-sequence diagrams.

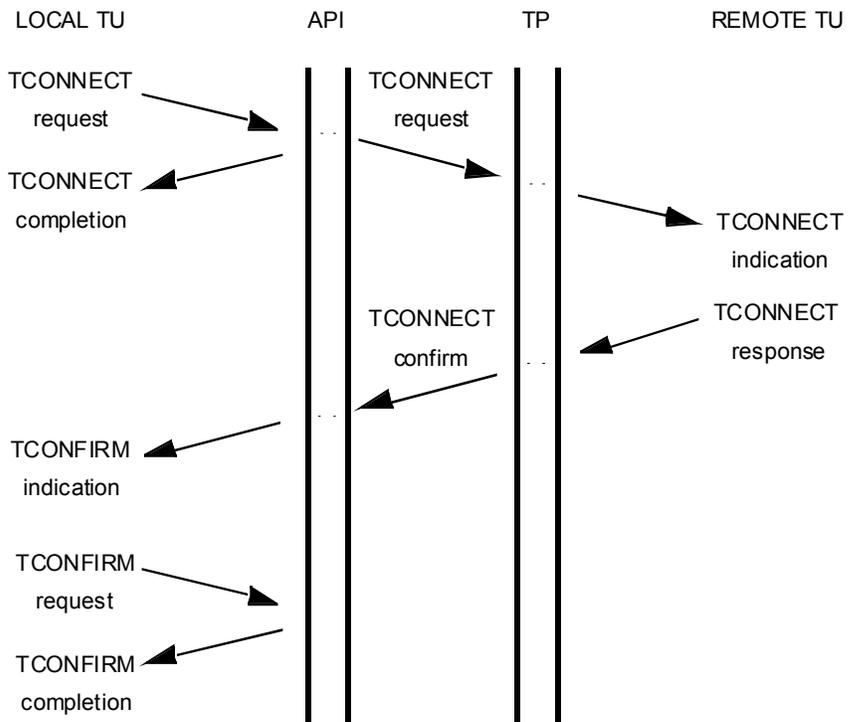
Local Endpoint Management (Initialization)



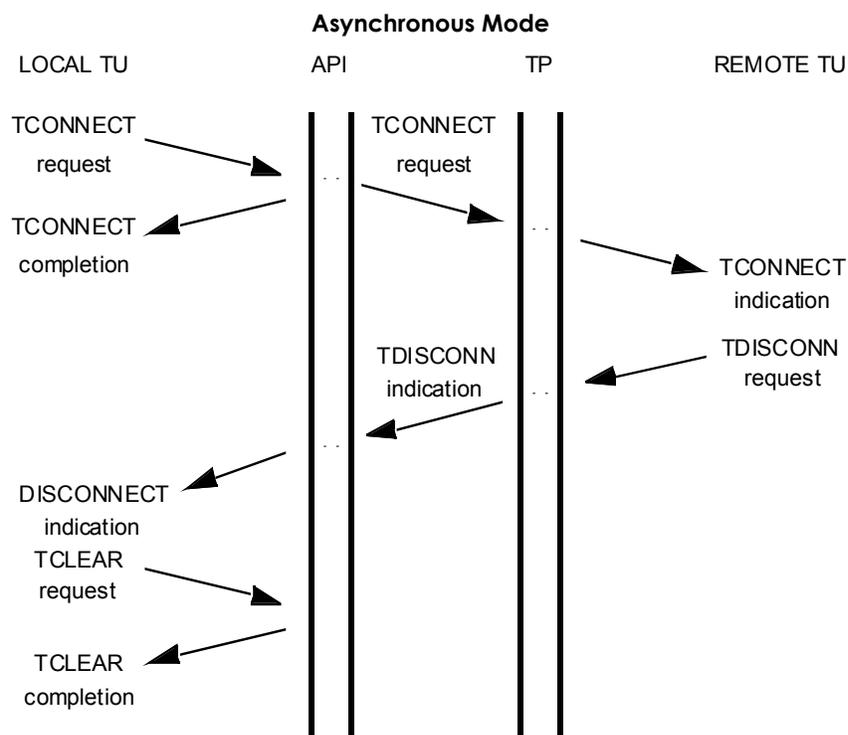
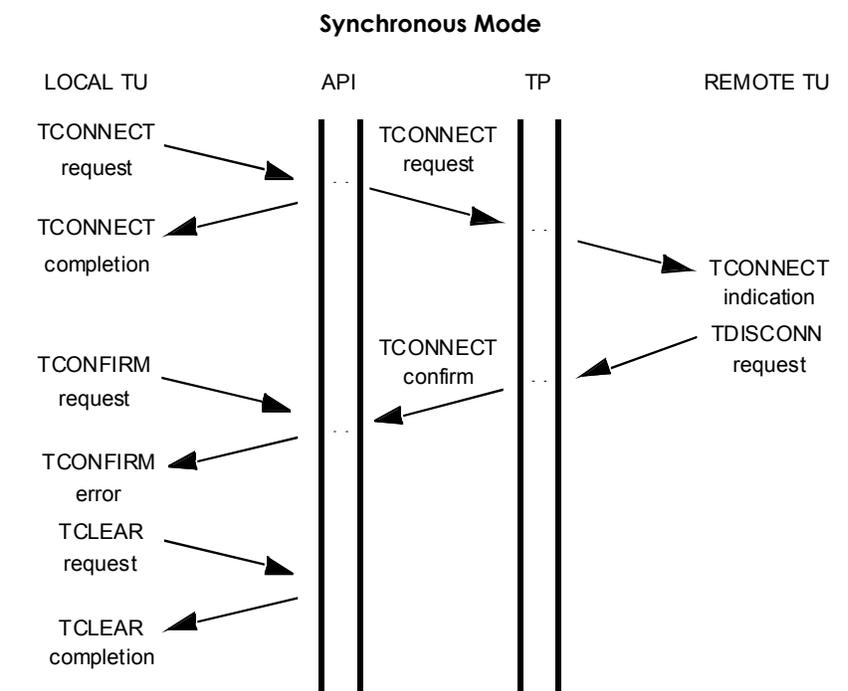
Client Connect Sequence (Accepted) Synchronous Mode



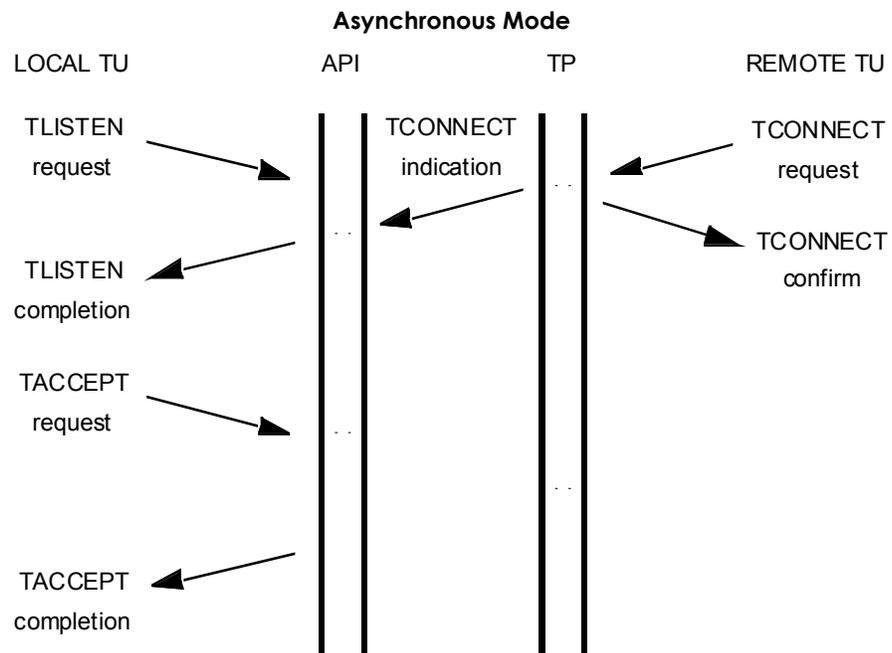
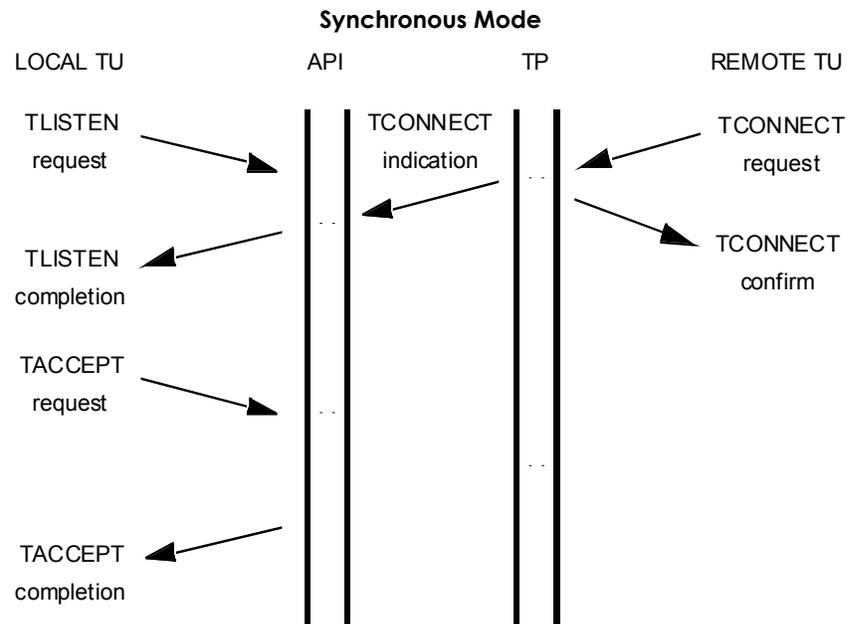
Asynchronous Mode



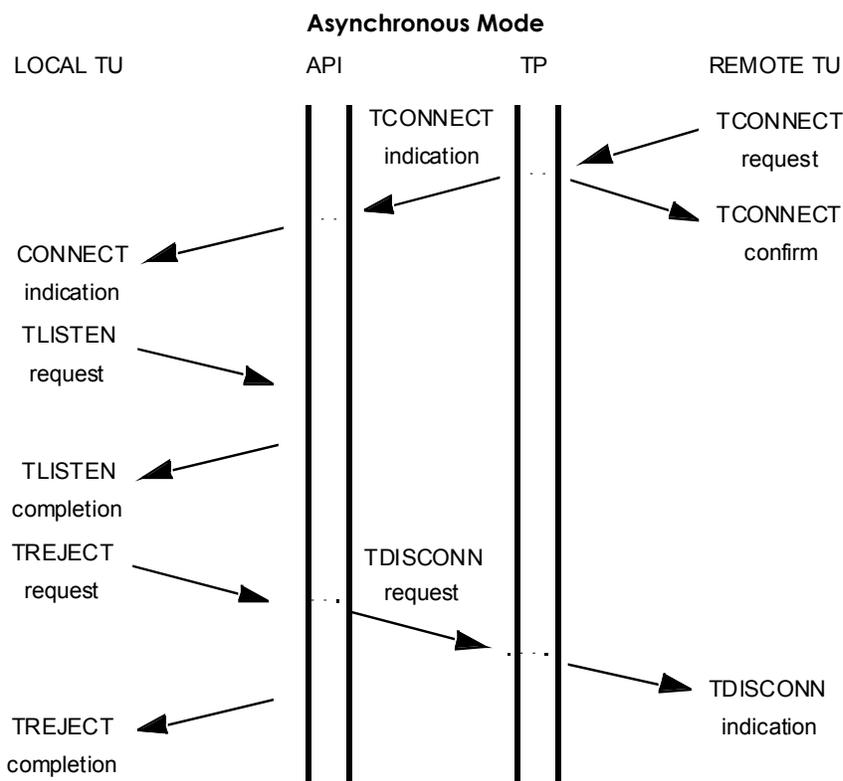
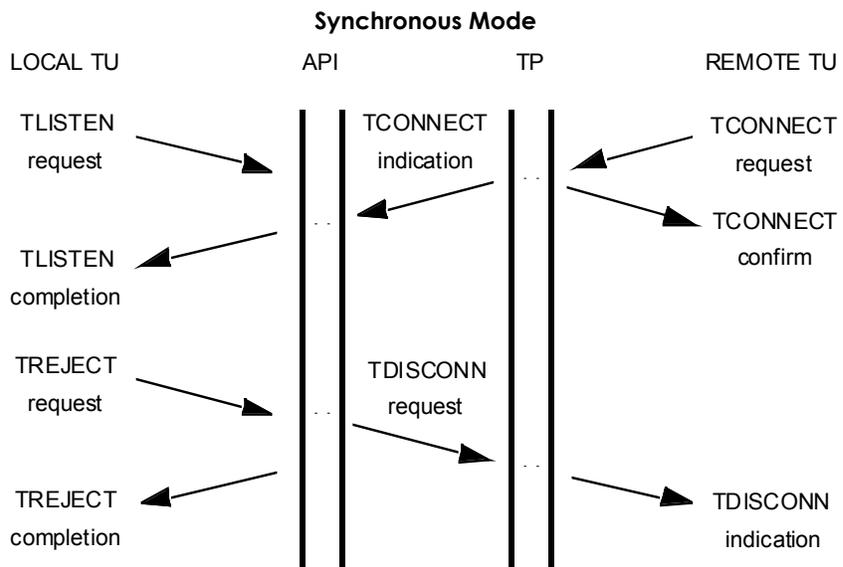
Client Connect Sequence (Rejected)



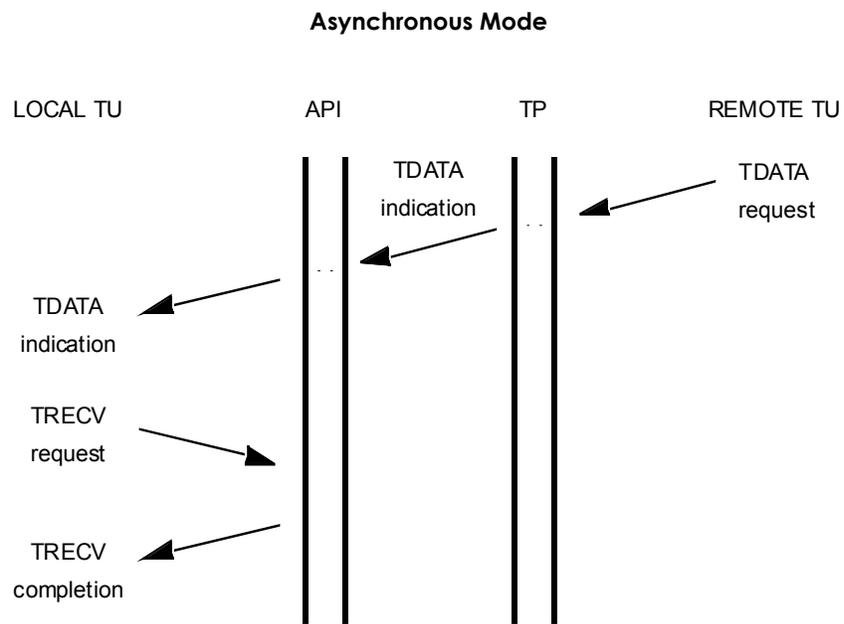
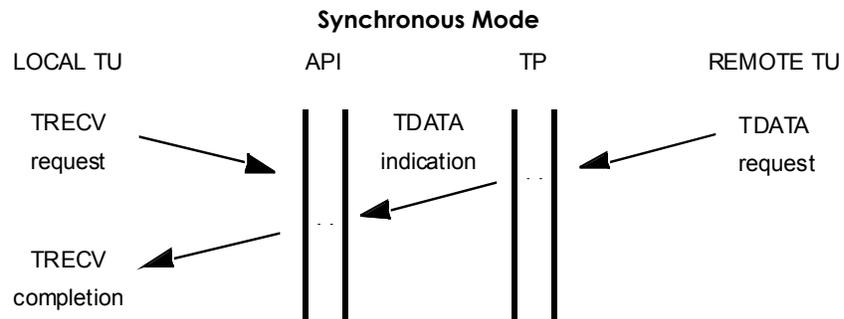
Server Connect Sequence (Accepted)



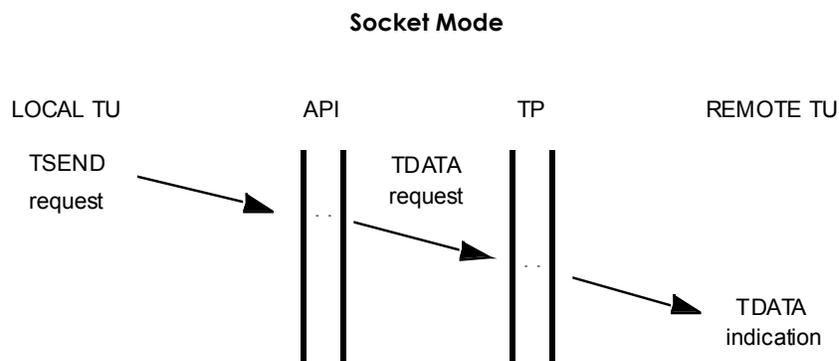
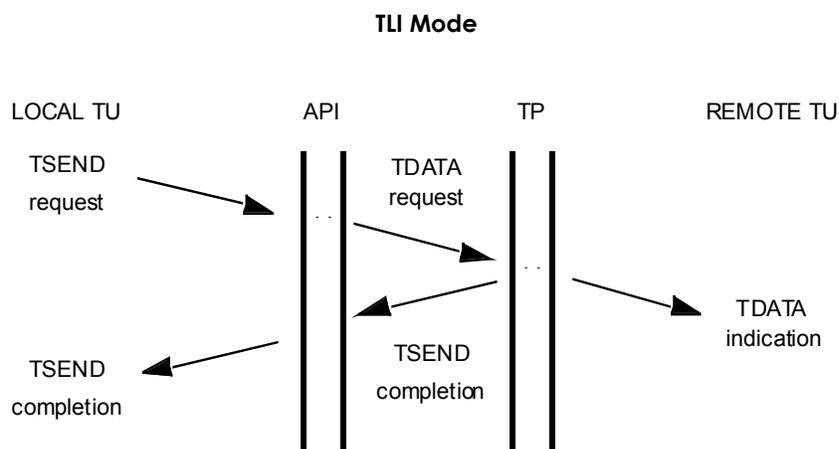
Server Connect Sequence (Rejected)



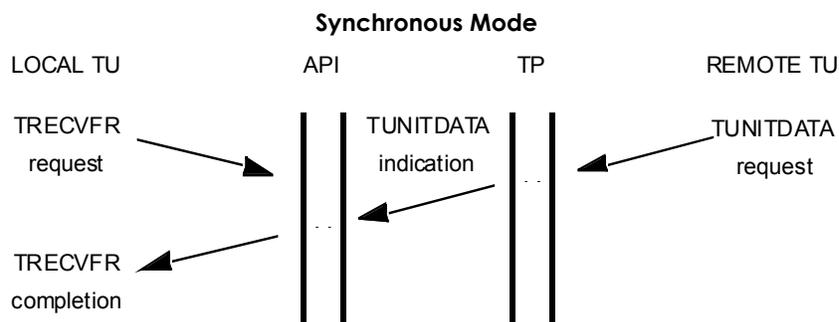
COTS Receive Data Sequence



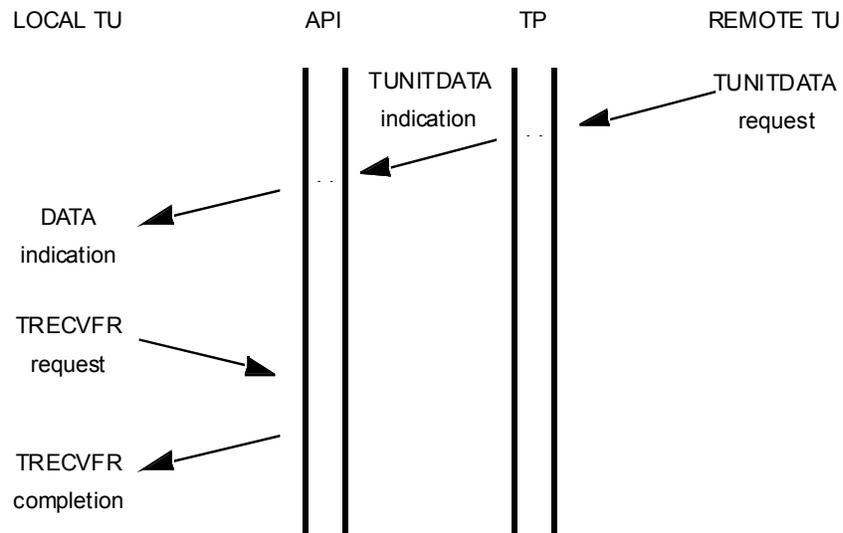
COTS Send Data Sequence



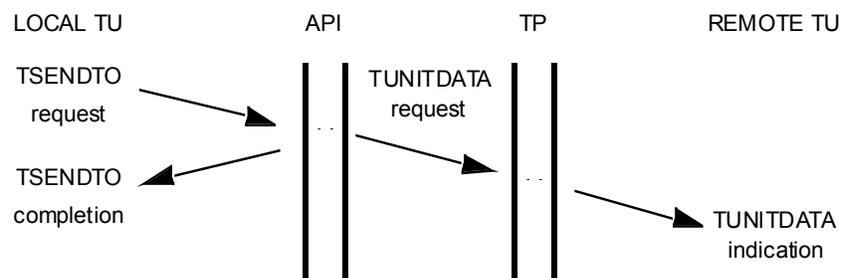
CLTS Receive Data Sequence



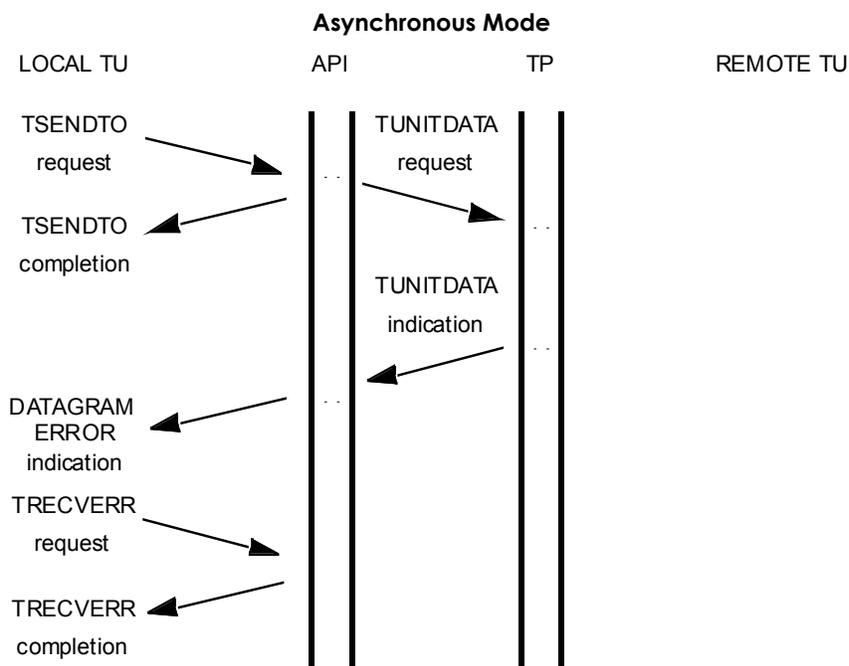
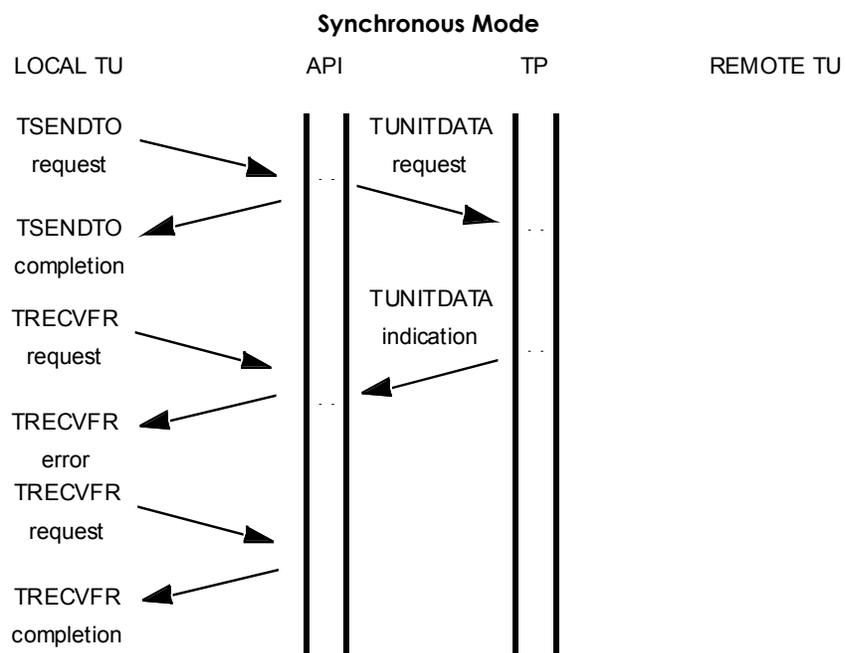
Asynchronous Mode



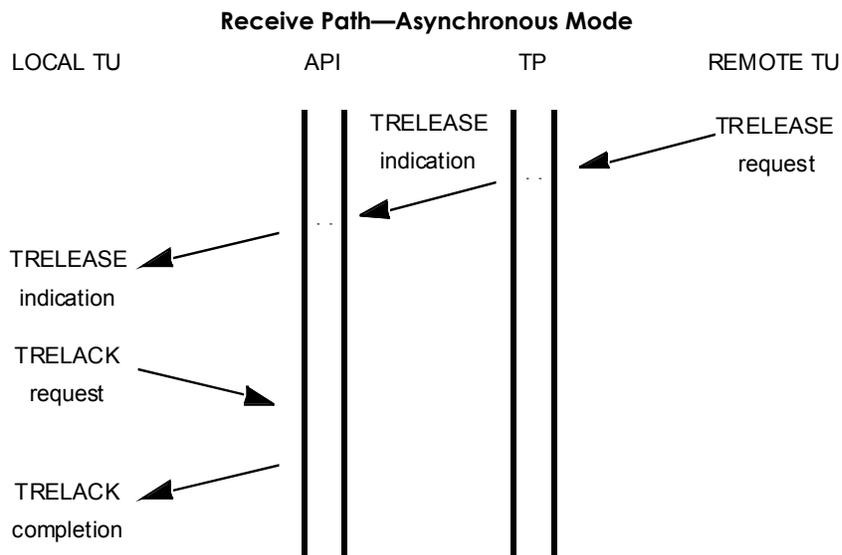
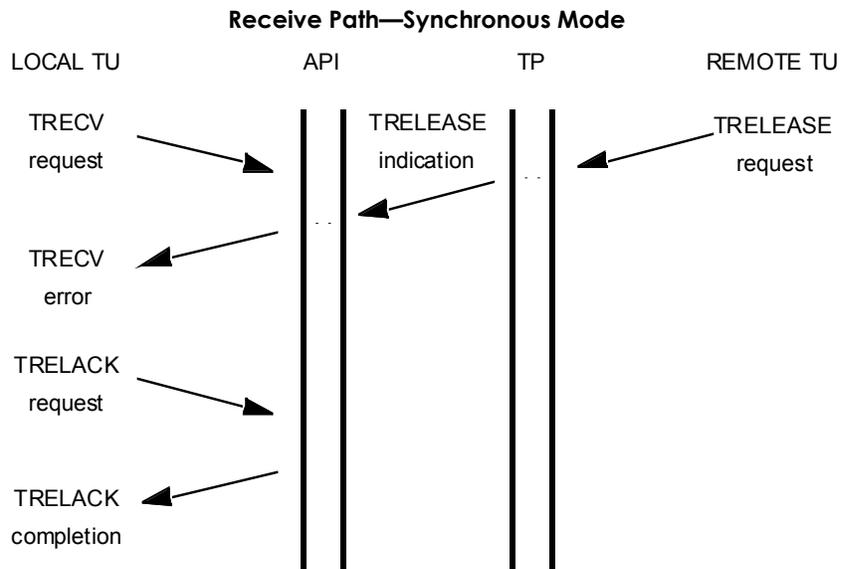
CLTS Send Data Sequence



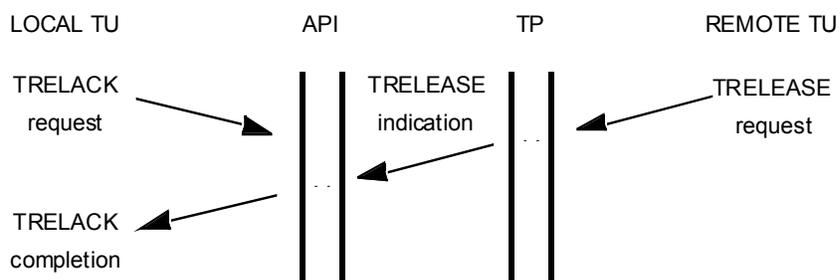
CLTS Datagram Error Sequence



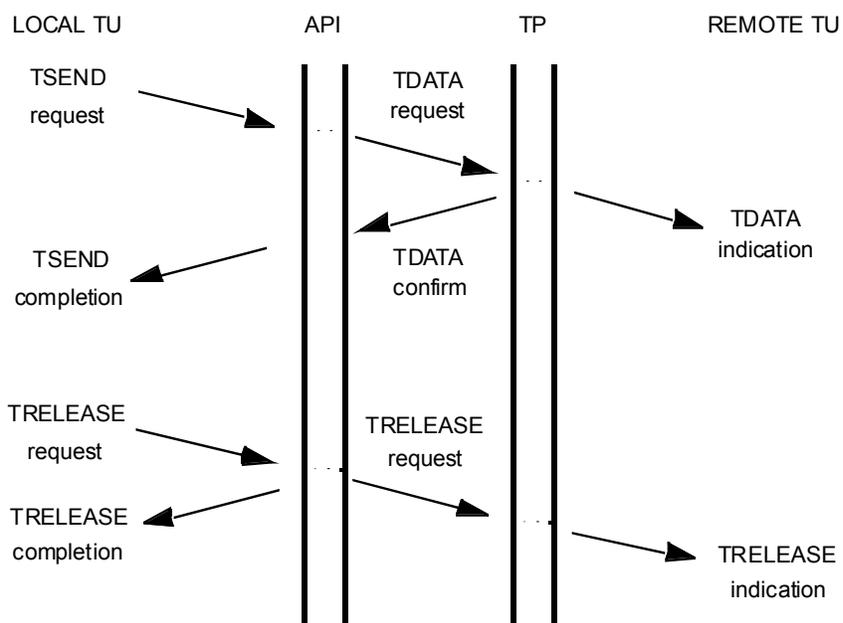
Orderly Release Sequence



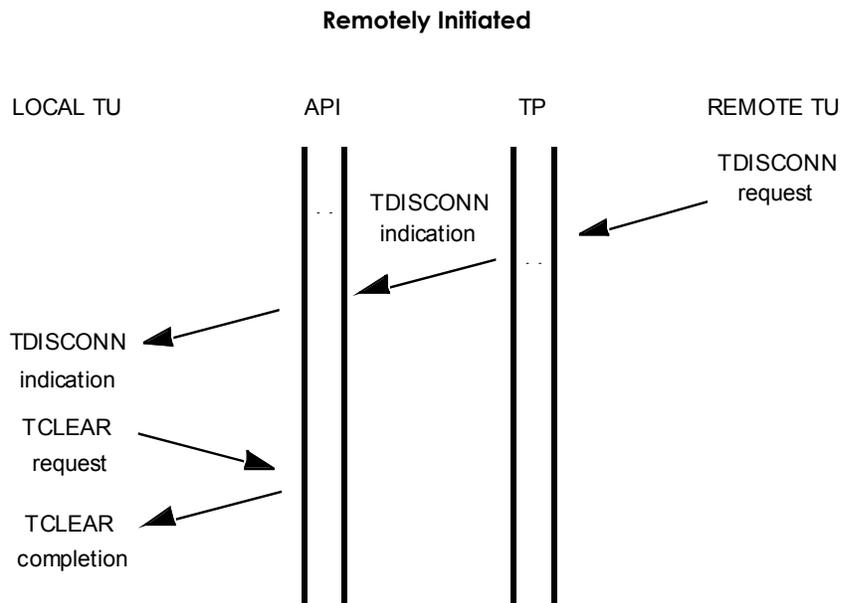
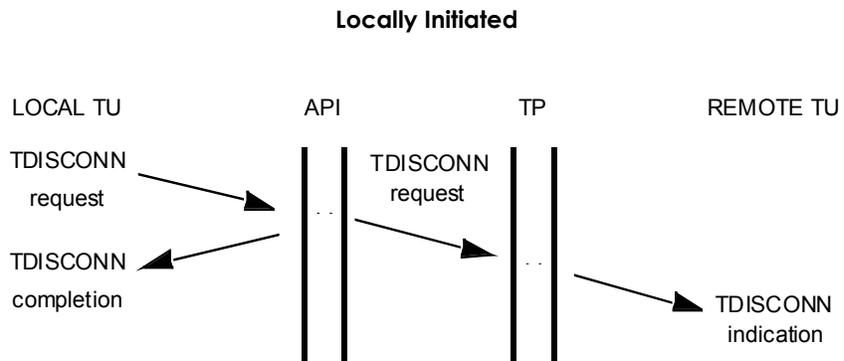
No Receive Data—Synchronous Mode



Send Path

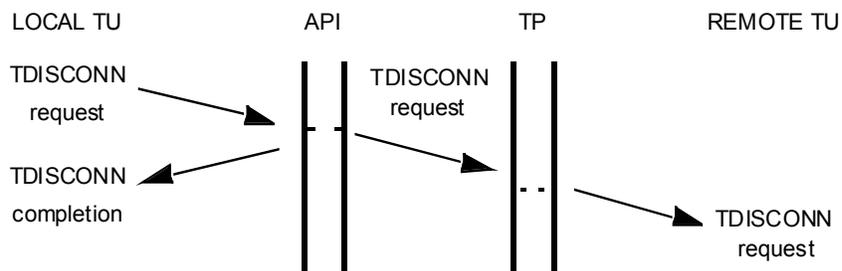


Abortive Disconnect Sequence

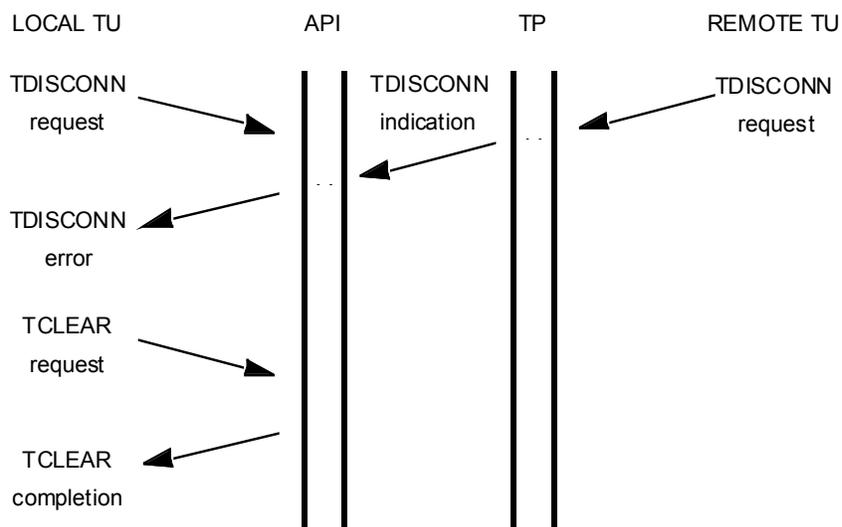


Simultaneous Disconnects

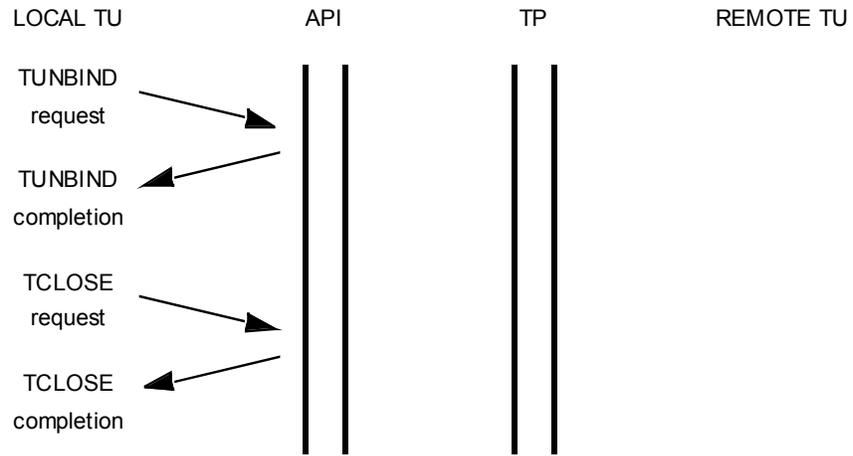
Transparent



Non-Transparent



Local Endpoint Management (Termination)



Using TCP and UDP Services

This appendix describes API considerations when using TCP or UDP as the underlying transport provider.

The following topics are discussed:

- [Protocol Address](#) – Describes the protocol address structure, transport layer address/port numbers, and network layer addressing/IP address
- [Expedited Data](#) – Describes how Unicenter TCPaccess maps TCP *urgent data* into API expedited data, including sending and receiving expedited data
- [Disconnect Reason Codes](#) – Lists the disconnect reason codes generated when a disconnect indicated is presented to the transport user application program
- [API-Initiated Protocol Actions](#) – Describes the actions performed by the TCP transport provider on behalf of the respective transport user API requests
- [Protocol Events Resulting In API Activity](#) – Describes the protocol events that result in API activity

Protocol Address

The protocol address structure used by the API consists of a domain type followed by a protocol-dependent protocol address. This structure is mapped by the macro APIDTPA for the assembler API and the tpairnet structure within the header api.h for the basic C library API.

The domain type portion of the protocol address used for TCP and UDP is INET (for Internet). The protocol-dependent portion of the INET domain type is the concatenation of the 16-bit TCP or UDP port number and the 32-bit IP network address.

Protocol dependencies apply to the structure of the Internet protocol address, the use of options, the use of API parameters, protocol-dependent disconnect reason codes, and the definition of expedited data. A summary of how protocol events and actions relate to API events and actions is also included.

Transport Layer Address—Port Numbers

In most applications using the client/server model, port numbers for server programs are defined by the application, and port numbers for client programs are determined by, or requested from, the transport provider prior to the sending of datagrams or request for connection.

Server programs typically use what are referred to as *well-known ports*. A well-known port is a specific port number that is always the same no matter when, where, or how the server program executes. This mechanism lets client programs reliably request connections to the desired server application independent of other factors. Use of well-known ports by the server and dynamic assignment of client port numbers also lend themselves to the one-to-many relationship that client/server applications use.

When selecting a port number for a server application, developers must have knowledge of the well-known port numbers used by other applications on their networks as well as *official* port numbers that are reserved for specific applications on an Internet-wide basis. Official well-known port numbers are used by common applications such as File Transfer Protocol (FTP) and Virtual Terminal (TELNET). These port numbers are in the range of 1-255 and are listed in the *Assigned Numbers RFC*.

If an application protocol having an officially assigned port is being implemented, the respective port numbers should be specified in the application's address structures. Some existing applications use well-known port numbers that are unofficially assigned. Where application-to-port number mappings are defined is implementation-dependent (for example, `file/etc/services` on UNIX-based systems; `SERVICE` statements in `APPCFGxx` for Unicenter TCPaccess).

Like most implementations of TCP and UDP, Unicenter TCPaccess divides the port number space between well-known (server) and client port numbers. In Unicenter TCPaccess, ports used as server ports are defined by the `PORTUSE` keyword parameter on the TCP and UDP parameter statements. The default range for `PORTUSE` is 1-4095. Ports used for clients ports are specified on the `PORTASGN` keyword parameter of the TCP and UDP parameter statements. The default port range for `PORTASGN` is 4096-65535.

Note: These are general conventions and exceptions are possible.

Local port numbers for TCP and UDP can be selected by the API application program by supplying the desired port number in the protocol address structure referenced by a `TBIND` request specifying `OPTCD=USE`. Local port numbers selected by the application must be less than 4096. Specifying local port numbers 4096 or greater conflicts with dynamic port assignment.

Applications request local port number assignment by issuing the TBIND request OPTCD=ASSIGN parameter. If an address buffer was supplied on the TBIND request, the transport provider returns the assigned local port number in the address structure.

An application that is requesting a TCP connection, UDP association, or UDP sending datagrams to a remote application must provide the remote port number that is used by that application. For TCP connections and UDP associations, this information is provided in the address structure referenced by the TCONNECT request. With UDP datagrams, the remote port is supplied with the TSENDTO request.

Server applications using the API normally do not need to know client port numbers. However, the transport provider passes the remote protocol address to the application on completion of a TLISTEN request, if an address buffer is provided. The remote protocol address is also available at any time after the connection or association is established via the TADDR OPTCD=REMOTE function.

Network Layer Addressing—IP Address

Following the port number in the address structure is the IP network address of the host containing the specified port. The application program normally is not concerned with the local IP address. In the API, this information must be left as zero or truncated in the address structure when TBIND is issued. This is because it is possible for Unicenter TCPaccess to have network interfaces to multiple networks and thus have multiple local IP addresses. It is the responsibility of the routing algorithms in Unicenter TCPaccess to determine which local network interface and associated local IP address are used to transmit data. The selected local IP address is available to the application after the connection is established by issuing a TADDR OPTCD=LOCAL API request.

A remote IP address must be provided when connecting or sending datagrams to a remote computer. The address structure supplied on a TCONNECT request must contain the remote IP address when establishing a TCP connection or UDP association. When sending UDP datagrams without the use of a UDP association, an address structure containing the remote IP address is supplied with each datagram on a TSENDTO request.

The API applications operating as servers do not normally require the remote IP address. However, it can be obtained by providing an address structure with the TLISTEN request. It may also be retrieved while the TCP connection or UDP association is in effect with a TADDR OPTCD=REMOTE API request.

An application using UDP without associations must supply an address buffer on the TRECVR request if the remote source IP address is desired.

Expedited Data

Unicenter TCPaccess maps TCP *urgent data* into API expedited data. There is no corresponding mechanism in the UDP protocol and therefore expedited data is not supported by the API when using UDP.

Sending Expedited Data

When an application issues a TSEND OPTCD=EXPEDITE request, the transport provider sets the TCP header urgent flag and sets the TCP header urgent pointer to the TCP sequence number offset corresponding to the TCP sequence number of the data byte following the last byte of TSEND OPTCD=EXPEDITE data. These are set in each TCP segment until all of the expedited/urgent data is transmitted.

Receiving Expedited Data

When urgent TCP data arrives from the network, Unicenter TCPaccess notifies the transport user of expedited data. This is accomplished by setting the EXPEDITE bit in the TPL of any outstanding TRECVC request. If there are no outstanding TRECVC requests when urgent TCP data arrives, and the transport user has enabled an expedited data indication exit routine, the transport provider drives that exit. In either case, all subsequent TRECVC requests complete with the EXPEDITE bit set until all of the urgent TCP data is received by the transport user. The expedited data indication exit is also disabled until all of the expedited data is received by the transport user.

A TRECVC request that receives all remaining TCP urgent data receives only TCP urgent data, even if there is space for additional data in the receive buffer and more non-urgent TCP data has arrived at the transport provider.

Disconnect Reason Codes

When a disconnect indication is presented to the transport user application program, the program can determine the reason for the disconnect indication (and clear the indication) by issuing a TCLEAR request.

Symbolic
Representations and
Descriptions

The symbolic representations and descriptions of the disconnect reason codes are listed in the following table. The symbolic disconnect reason codes are mapped into numeric codes by the APIDTPL macro with the DOMAIN=INET parameter.

TDTRANTO	Excessive unacknowledged retransmissions of TCP data caused the local transport provider to consider the TCP connection disconnected. This condition is sometimes referred to as retransmission time-out.
TDHOSTUN	An ICMP <i>host unreachable</i> message was received. This may be due to a routing configuration problem or a failure of some network component necessary to reach the desired destination.
TDPORTUN	An ICMP <i>port unreachable</i> message was received from the remote host. The desired port is either inactive or unsupported.
TDRABORT	A TCP segment was received with the RST (reset) bit on in the TCP header. This may be the result of the remote TCP detecting a fatal error in the TCP connection or may have been initiated by the application that is using TCP.
TDLNIDWN	The local network interface necessary to reach the remote host is inactive.
TDPROTUN	An ICMP <i>protocol unreachable</i> message was received from the remote host. TCP is inactive or unsupported on the desired destination.
TDACPRR	The connection was terminated due to an unrecoverable error within some component of the ACP.
TDAPIRR	The connection was terminated due to an unrecoverable error within some component of the API.
TDNETUN	An ICMP <i>net unreachable</i> message was received.
TDNOFRAG	An ICMP <i>fragmentation needed and DF set</i> message was received.
TDSRFAIL	An ICMP <i>source route failed</i> message was received.

API-Initiated Protocol Actions

This section briefly describes the actions performed by the TCP transport provider on behalf of the respective transport user API requests. Only the API requests that cause some TCP-defined protocol activity are listed.

The following requests and protocol actions are **not** included:

- API requests that are processed by the transport provider but do not direct any action by TCP
- Protocol actions of UDP, because they reduce to the sending of data and, in the case of UDP associations, simulation of connection establishment and termination

Macros and
Associated
Protocol Actions

The following table lists the macros and the associated protocol actions.

Macro	Protocol Action
TCONNECT	Send initial SYN.
TDISCONN	Send TCP RST.
TRECV	Increase the available receive window space.
TREJECT	Send TCP RST.
TRELEASE	Send TCP FIN.
TSEND	Make data available for packetization; cause immediate transmission with PUSH bit set if OPTCD=NOMORE was in effect. TSEND requests are completed when all of the data sent is acknowledged by the remote TCP.

Protocol Events Resulting In API Activity

This section lists the protocol events that result in API activity.

Initial SYN Arrives (TCP)

If there is a TLISTEN pending completion within the provider, it is completed. If there was no pending TLISTEN, but the transport user provided a connect indication exit, the exit is driven unless the exit was previously driven and the transport user had not issued TLISTENs for all of the outstanding connect indications.

SYN/ACK Arrives in Response to a Previously Sent Initial SYN(TCP)

If there is a TCONFIRM pending completion within the provider, it is completed. If there was no pending TCONFIRM, but the transport user provided a connect confirmation exit, the exit is driven.

Data Arrives (TCP/UDP)

If there are any pending TRECvs or TRECvFRs awaiting the arrival of data, completion occurs until there is no more data or no more requests. If there were no pending TRECvs or TRECvFRs, and the transport user provided a data indication exit, that exit is driven. If UDP associations are in use, the first datagram to arrive is also considered a connect indication and is processed in the same manner as the arrival of an initial TCP SYN.

Acknowledgment for Sent Data Arrives (TCP)

The associated pending TSEND requests are completed. If the endpoint is operating in socket mode, the Send Window Open event is raised if the window was previously closed and there are no pending TSEND requests.

Urgent Data Arrives (TCP)

If there are no pending TRECvs awaiting data and the transport user provided an expedited data indication exit, it is driven. The endpoint enters urgent mode. All TRECv requests complete with the EXPEDITE bit set in the TPL while in urgent mode.

ICMP Message Arrives

Any ICMP message that indicates a permanent error causes a disconnect indication to be generated. The disconnect indication is presented to the transport user via the completion of pending requests, if any. If not, the disconnect indication exit is driven, if it exists. Any subsequent connection termination request (except TCLEAR unless OPTCD=CLEAR was in effect) or data transfer request completes abnormally until the disconnect indication is cleared. In any case, the respective disconnect reason code is set within the endpoint and is available to the transport user via TCLEAR.

A TCP RESET Arrives

A disconnect indication is passed to the transport user and processed according to the same method that is used when a permanent ICMP error occurs.

A FIN Arrives (TCP)

If there is a TRELACK pending at the endpoint, it can be completed after all outstanding TRECvs have completed. Any outstanding or subsequently issued TRECv for which there is no data is completed with a release indication in the return code. If the release indication is not presented in a TRECv request and the transport user provided a release indication exit, the exit is driven after all TRECvs that complete normally (that is, with data) have completed.

API Data Sets

This appendix lists the data sets used by the API and describes their usage.

Data Sets

This list describes the data sets used by the API.

MAC	Contains all assembler language macro definitions for the API. Should be included in SYSLIB DD statement for all application program assembly steps that reference API macro instructions.
H	Contains all C language include files used by the API. Should be included in the SYSLIB DD statement that defines the location of include files used with the IBM C/370 and SAS/C compilers. Note: Required when compiling any application program that references C or socket library routines.
CILIB	Contains the C and socket library routines in load module form. Should be included in the linkage editor SYSLIB DD statement when building nonreentrant application programs compiled with the IBM C/370 compiler.
CSLIB	Contains the C and socket library routines in load module form. Should be included in the linkage editor SYSLIB DD statement when building nonreentrant application programs compiled with the SAS/C compiler.
CIROBJ	Contains the C and socket library routines in object module form. Should be included in the SYSLIB DD statement for the pre-link step when building reentrant application programs compiled with the IBM C/370 compiler.

- CSROBJ Contains the C and socket library routines in object module form. Should be included in the SYSLIB DD statement for the pre-link step when building reentrant application programs compiled with the SAS/C compiler.
- PARM Contains configuration members used to initialize TCPaccess, the API, and the NDS when these task groups begin execution.
Note: Can be modified during installation to customize the configuration of the API.
- SAMP Contains the sockcfg.c (SOCKCFG) source member that defines configuration information used by socket library routines.
Note: Can be modified and compiled at installation time to customize the configuration of the socket library.

Sample Assembler Program

This appendix describes the sample assembler program, TTCP, included with your Unicenter TCPaccess software.

Sample Program

A sample assembler application program is included in the Unicenter TCPaccess SAMP library. You can print the program, or you can browse or edit it online. If you plan to use the sample program as a template, you should first copy it so you do not destroy the source.

The sample program includes the source for the TTCP TSO command processor, a test program for exercising Unicenter TCPaccess and the API functions. It is based on the UNIX tcp programs. Information on its usage is given in the *User Guide*.

The program is simple and is extensively commented to assist a prospective API programmer in designing and coding an application program.

The program consists of these members:

TTCCPCP	Command processor front end to TTCPR and TTCPT routines.
TTCPMSGs	Contains TTCP message definitions.
TTCPR	TTCP receiver listens for a connection request and attaches a processing subtask part of the TTCP receiver operation.
TTCPRST	TTCP receiver subtask receives data using overlapped TRECvs with ECB posting on completion.
TTCPT	TTCP transmitter sends data using overlapped TSENDs with ECB posting on TSEND completion.

Index

2

24-bit addressing[*twenty-four*], 3-55

3

31-bit addressing[*thirty-one*], 3-55

A

abortive disconnect sequence, B-14

access points
 binding, 2-6
 unbinding, 2-6

ACLOSE macro, 2-25

addresses, binding and unbinding, 2-34

addressing
 24-bit, 3-55
 31-bit, 3-55

AOPEN macro, 2-24

APCB macro, 2-71

API
 acknowledgment for sent data arrives, C-7
 concepts and terminology, 2-5
 connectionless-mode service, 2-5
 connection-mode service, 2-5
 data arrives (TCP/UDP), C-7
 data sets, D-1
 exit routines, 3-18
 FIN arrives (TCP), C-8
 ICMP message arrives, C-8
 initial SYN arrives (TCP), C-7
 macro parameters, C-6

 multitasking, 3-54
 protocol
 address structure, C-1
 domain type, C-1
 events, C-7
 relationship independence, 2-2
 service request modes, 2-3
 service requests, 2-3
 SYN/ACK arrives, C-7
 TCP RESET arrives, C-8
 transport service access point, 2-6

API activity, C-7

API, address structure, C-1

APIDTPL macro, C-5

API-initiated protocol actions, C-6

application layer protocols
 FTP, 1-9
 SMTP, 1-9
 TELNET, 1-9

Application Program Control Block (APCB), 2-24

application programs
 example, 2-25
 organization, 2-23, 3-53

ARPANET definition, 1-4

assembler
 sample application program, E-1
 TTCPCMSG, E-1
 sample application program members
 TTCPCP, E-1
 TTCPR, E-1
 TTCPRST, E-1
 TTCPT, E-1

ASSOC service, 2-64

asynchronous
 exit routines, 3-23
 coding procedures, 3-24
 example, 3-25

- register information, 3-24
- mode, B-2
- operation, 3-9
 - Event Control Block (ECB), 3-10
 - internal ECB, 3-11
 - using external ECB, 3-13

B

Berkeley Software Distribution (BSD), 1-4

binds

- addresses, 2-34
- available address, 2-34
- listen queue relationship, 2-36
- to well-known address, 2-35

BSD, Berkeley Software Distribution, 1-4

C

calls to synchronous exit routines, 3-20

checking for orderly release, 2-56

client, 2-7

client mode, 2-7

client/server connections, 2-9

close endpoints, 2-29, 2-32

CLTS

- data transfer functions, 2-60
- datagram error sequence, B-11
- overview, 1-24
- receive data sequence, B-9
- send data sequence, B-10

codes

- diagnostic, 3-51
- disconnect reason, C-5
- general return, 3-48
- recovery action, 3-50
- register, 3-48
- TPL return codes, 3-48

common parameters, 2-13

completion events, B-2

confirm primitive, 2-3

connection mode, 1-14

connectionless mode, 1-14

- connectionless-mode service, 2-57
 - local endpoint management, 2-57
 - states, 2-78
 - with associations, 2-63

- connection-mode service, 2-27
 - miscellaneous functions, 2-38
 - phases, 2-27
 - states, 2-76

Connection-Oriented Transport Service. See COTS., 1-14

connections

- client, 2-7
- client/server, 2-9
- establishment functions, 2-42
- establishment of, 2-27, 2-42, 2-55
- initiation of, 2-43
- listening for, 2-66
- long-term binding, 2-6
- release, 2-27
- release functions, 2-53
- server, 2-7
- server mode, 2-8

context for the language environment, 3-46

context related to the transport user, 3-46

COTS

- data transfer functions, 2-48
- endpoint states, 3-35
- endpoints, 2-6
- receive data sequence, B-8
- send data sequence, B-9
- service primitives, 1-15
- services
 - data transfer, 1-14
 - transport connection establishment, 1-14
 - transport connection release, 1-14

D

DARPA

- addresses, 1-26
- datagram service, 1-24
- definition, 1-4
- Domain Name System (DNS), 1-29
- protocol suite, 1-9

- data transfer, 2-27, 2-48, 2-60
 - connection release, 2-53
 - service functions for, 2-60

data transfer phase, 2-8

datagram
 outgoing, 2-61
 receiving, 2-62
 UPD, C-3

DDN, Defense Data Network, 1-4

declarative macro instructions, 2-70

Defense Data Network (DDN), definition, 1-4

destination transport user, 1-29

diagnostic codes, 3-51

directory service, 1-29

disconnect reason codes, symbolic representation, C-5

Disconnect Service, 2-11

DNS definition, 1-29

Domain Name System. *DNS*

domain type
 protocol address structure, C-1

DOMAIN=INET parameter, C-5

E

ECB, 3-10
 exit routines, 3-11
 internal, 3-11

end of transport, 2-50

endpoint states
 connected, 3-35
 connect-indication-pending, 3-35
 connect-in-progress, 3-35
 [defined](#), A-1
 release-indication-pending, 3-35
 release-in-progress, 3-35
 RSOURLSE, A-6
 summary table, A-1
 TSCONNECT, A-6
 TSENABLED, A-4
 TSINCONN, A-4
 TSINRLSE, A-6
 TSOUCONN, A-4

endpoint, state variables for, A-1

endpoints
 binding to client endpoint, 2-34
 binds to well-known address, 2-35

 bound to local transport address, 2-43
 changing user IDs, 2-41
 closing, 2-32
 functions, 2-73
 listening, 2-46
 management, 2-57
 opening and closing, 2-29
 opening of, 2-58
 passing control of, 2-32, 2-33
 specifying user IDs, 2-41
 state transitions, 2-75
 states, 2-73
 termination of, 2-59
 TU end of connection, 2-6
 unbinding with TUNBIND, 2-36

EOM/NOTEOM option code, 2-50

error
 events, B-2
 recovery, 2-26

error handling, 3-9
 ACLOSE errors, 3-52
 AOPEN errors, 3-52
 asynchronous, 3-47
 diagnostic codes, 3-51
 general return codes, 3-48
 macro information, 3-47
 recovery action codes, 3-50
 register codes, 3-48
 specific error codes, 3-51
 synchronous, 3-47
 TPL return codes, 3-48

establishment of
 associations, 2-65
 connections, 2-42
 sessions, 2-23

Ethernet, 1-4

examples
 asynchronous exit routine, 3-25
 synchronous exit routine, 3-22

executions, fully initialized TPL, 2-69

exit routines, 2-31
 APEND, 3-44
 asynchronous, 3-23
 coding procedures, 3-24
 example, 3-25
 register information, 3-24
 CONFIRM, 3-33
 CONNECT, 3-32
 DATA, 3-33

- deriving context in, 3-46
- DGERR, 3-34
- DISCONN, 3-35
- LERAD, 3-38, 3-40
- parameter list, 3-26
- RELEASE, 3-35
- scheduling DATA and XDATA, 3-37
- specifying exit routines, 3-18
- summary, 3-27
- SYNAD, 3-38, 3-41
- SYNAD/LERAD errors, 3-39
- synchronous
 - calling exit routines, 3-20
 - coding procedures, 3-21
 - example, 3-22
 - register information, 3-20
- synchronous error recovery, 3-37
- TEXTST, 3-19
- TPEND, 3-19, 3-42
- TPL, 3-18
- transport endpoint exit parameters, 3-26
- TXP information, 3-26
- using exit routines, 3-27
- XDATA, 3-34, 3-36

exits

- AOPEN exit lists, 3-19
- calling exit routines. See exit routines., 3-20
- protocol events, 3-31
- TEXTST, 3-18
- TOPEN exit lists, 3-19
- TPL completion, 3-30

expedited data

- receiving, C-4
- sending, C-4

F

File Transfer Protocol. *FTP*

fixed-length parameters, 2-14

FTP, File Transfer Protocol, 1-9

functions

- connection establishment, 2-42
- connection release, 2-53
- COTS data transfer, 2-48
- data transfer service, 2-60
- miscellaneous, 2-38
- TACCEPT, 2-45
- TCHECK, 2-68
- TCLEAR, 2-54

- TCONNECT, 2-43
- TDISCONN, 2-54, 2-55
- TERROR, 2-69
- TEXEC, 2-69
- TOPEN, 2-58
- TREJECT, 2-45
- TRELACK, 2-56
- TRELEASE, 2-55
- TRETRACT, 2-47
- TUNBIND, 2-59
- TUUNBIND, 2-36

I

ICMP message arrives, C-8

indications

- additional data, 2-50
- type of data, 2-51

initiations

- abortive release, 2-54
- connection, 2-43

Internet protocol address, C-1

invocation of asynchronous exit routines, 3-23

ISO standards, ISO 8072, 1-13

L

listen queue, 2-34

listening

- endpoint, 2-46
- for connect indications, 2-66

local endpoint

- control processing, 2-67
- management, 2-27
- management (termination), B-16

logger service, 1-29

M

machine-readable endpoint state transition table, A-1

macro instructions

- ACLOSE, 2-25
- AOPEN, 2-24, 3-19
- APCB, 2-71

- associated protocol actions, C-6
- EXIT operand, 3-14, 3-18
- OPTCD=SYNC, 3-8
- synchronization characteristics, 3-6
- TACCEPT, 3-32
- TCHECK, 3-10
- TCHECK in TPL exits, 3-30
- TCLOSE, 2-32
- TCONFIRM, 3-33
- TDSECT, 2-71, 3-26
- TEXTST, 2-71, 2-72, 3-19
- TLISTEN, 3-32
- TOPEN, 2-30, 3-19
- TPL, 2-72
- TRECV, 3-9, 3-36
- TREJECT, 3-32
- TRELEASE, 3-35

manipulation of protocol options, 2-39

modes of operation

- asynchronous operation, 3-9
 - ECBs, 3-11
 - using external ECB, 3-13
 - using internal ECB, 3-11
- synchronization characteristics, 3-6
- synchronous operation, 3-8
 - error handling, 3-9
 - operation flow, 3-8

MORE/NOMORE option code, 2-50

multi-address-space, 3-2

multiple address spaces, 3-55

multitasking

- in an API environment, 3-54
- programs, 3-2
- rules for, 3-54

multithreaded

- programs, 3-2
- servers, 2-46

N

- network address, 2-15
- network layer, addressing IP address, C-3
- NORMAL/EXPEDITE option code, 2-51

O

- obtaining basic protocol information, 2-38
- Open Systems Interconnection. See OSI, 1-1
- opens endpoints, 2-29, 2-58
- orderly release
 - procedure, 2-55
 - sequence, B-12
- Orderly Release Service, 2-11
- OSI
 - connections
 - connection endpoint, 1-12
 - transport endpoint, 1-12
 - Reference Model, 1-5
 - application layer protocols, 1-9
 - layer interaction, 1-6
 - service access points, 1-11
 - service data units (SDU), 1-12
 - Transport Service Modes
 - connectionless-mode protocols, 1-3
 - connection-mode protocols, 1-3
 - transport services and protocols, 1-11

P

- parameter lists
 - in-line, 2-12
 - listforms, 2-12
 - transport endpoint exit parameters, 3-26
 - TXP information, 3-26
- parameters
 - common to all requests, 2-13
 - fixed-length, 2-14
 - variable-length, 2-14, 2-21, 2-22
- partial protocol address, 2-15
- passing control of an endpoint, 2-32, 2-33
- primitives
 - T.confirm, 3-31
 - T.indication, 3-31
 - T-DISCONNECT.request, 3-7
 - T-EXPEDITED-DATA.indication, 3-36
- process event indications, 2-26
- processing service requests, 2-3
- protocol
 - actions initiated by API, C-6

- address
 - domain type, C-1
 - network layer IP address, C-3
 - partial, 2-15
 - structure, C-1
 - transport layer port numbers, C-2
- event exits, 3-31
- protocol-dependent disconnect reason codes, C-1
- protocols
 - actions associated with API macros, C-6
 - application layer protocols, 1-9
 - events resulting in API activity, C-7
 - IEEE 802.X series, 1-10
 - manipulation of options, 2-39

R

- reason codes, C-1
- receives
 - connect indication, 2-44
 - expedited data, C-4
 - incoming datagrams, 2-62
- recovery action codes, 3-50
- rejection, connection indication, 2-45
- relationship Independence, 2-2
- requesting association-mode service, 2-64
- response primitive, 2-3
- retraction of listen request, 2-47
- retrieve addresses, 2-59

S

- sample assembler application program, E-1
- SDU, service data units, 1-12
- send
 - expedited data, C-4
 - outgoing datagrams, 2-61
- server, 2-7
- server mode
 - multithreaded servers, 2-8
 - single-threaded servers, 2-8
- service

- functions
 - TCONNECT, 2-65
 - TINFO, 2-38
 - TLISTEN, 2-66
 - TRECVFR, 2-62
 - TSENDTO, 2-61
 - TUSER, 2-41
- parameters, 2-11
- request modes
 - asynchronous, 2-3
 - synchronous, 2-3
- request processing, 2-3
- requests, 2-11, 2-69
- Service Data Units (SDU), 1-12
- setting the TPL inactive, 3-17
- Simple Mail Transfer Protocol. See SMTP., 1-9
- simultaneous disconnects, B-15
- single-threaded servers, 2-44
- SMTP, simple mail transfer protocol, 1-9
- state
 - transition tables, A-2
 - transitions, 2-75, 2-76
 - transitions for connection-mode service, 2-76
 - variable, A-1
- synchronization characteristics
 - of macro instructions, 3-6
- synchronization modes, 3-17
- synchronous
 - error recovery
 - exit routines, 3-37
 - SYNAD/LERAD errors, 3-39
 - exit routines, 3-20
 - coding procedures, 3-21
 - example, 3-22
 - register information, 3-20
 - mode, B-2
 - operation, 3-8
 - error handling, 3-9
 - operation flow, 3-8

T

- TACCEPT function, acceptance of connect indication, 2-45
- TADDR service, 2-59

task synchronization, 3-2

TBIND macro, 2-34

TCHECK control function, 2-68

TCLEAR
 function, 2-55
 return disconnect information, 2-54

TCLOSE macro, 2-32

TCONNECT
 function, 2-43
 service function, 2-65

TCP
 connection model, 1-18
 data arrives (TCP/UDP), C-7
 data transfer model, 1-20
 RESET, C-8
 Transmission Control Protocol, 1-10
 transport service, 1-21

TDISCONN function, 2-54, 2-55

TDSECT macro, 2-71

TELNET, 1-9

termination, connectionless-mode endpoint, 2-59

TERROR control function, 2-69

TEXEC control function, 2-69

TEXTST
 exit routines, 3-18, 3-19
 macro, 2-71, 2-72

TIB, 2-39

time-sequence diagram
 abortive disconnect sequence, B-14
 client connect sequence (accepted), B-4
 client connect sequence (rejected), B-5
 CLTS
 datagram error sequence, B-11
 receive data sequence, B-9
 send data sequence, B-10
 COTS
 receive data sequence, B-8
 send data sequence, B-9
 initialization, B-3
 local endpoint management, B-3
 local endpoint management (termination), B-16
 orderly release sequence, B-12
 server connect sequence (accepted), B-6
 server connect sequence (rejected), B-7
 simultaneous disconnects, B-15

TINFO service function, 2-38

TLISTEN
 function, 2-44
 service function, 2-66

token bus, 1-4

token ring, 1-4

TOPEN
 function, 2-58
 macro, 2-30

TOPTION service, 2-39

TP, 2-2

TPL
 common prefix, 2-19
 completion exit, 3-30
 exit routines, 3-11
 return codes, 3-48

TPL macro, 2-72

Transmission Control Protocol (TCP), 1-10

transmitting data, C-3

transport
 address, 2-15
 endpoint
 exit parameters, 3-26
 interface, B-1
 data unit, 2-53
 protocols
 transport service provider, 1-11
 transport service user, 1-11
 provider, 2-23, B-1
 service
 access point, 2-6
 data unit, 2-53
 parameter list, 2-11, 2-17
 provider, 2-2
 user, 2-2
 services
 transport service provider, 1-11
 transport service user, 1-11
 user, 2-7
 user data, 2-15
 expedited data, 2-49
 normal data, 2-49

Transport Information Control Block(TIB), 2-39

transport layer
 address <bul> port numbers, C-2
 addressing, 1-26
 destination transport user, 1-29

directory service, 1-29
services
 connection mode, 1-14
 connectionless mode, 1-14
 Connectionless Transport Service (CLTS),
 1-24
 Data Transfer, 1-18
 expedited data transfer service, 1-20
 TCP data transfer model, 1-20
 endpoint states and service sequence, 1-22
 logger service, 1-29
 network classes, 1-28
 Transport Connection Establishment, 1-16,
 1-18
 Transport Connection Release, 1-21
 user datagram protocol (UDP), 1-10
Transport Service Access Points (TSAP), B-1
TRECVFR service function, 2-62
TREJECT function, 2-45
TRELACK function, 2-56
TRELEASE function, 2-55
TRETRACT function, 2-47
TSAP, transport service access point, B-1
TSDU, transport service data unit, 1-20
TSENDTO service function, 2-61
TTCPCP, sample application program member, E-1
TTCPPMSGs sample application program member,
E-1
TTCPR sample application program member, E-1

TTCPRST sample application program member, E-1
TTCPT sample application program member, E-1
TU, 2-2
TUNBIND
 function, 2-59
TUNBIND function, 2-36
TUSER service function, 2-41

U

UDP datagrams, C-3
unbind
 addresses, 2-34
 endpoints, 2-36
user data
 length, 2-53
 parameter, 2-53
User Datagram Protocol (UDP), 1-10

V

variable-length parameters, 2-21, 2-22
 transport protocol address, 2-14
 transport protocol options, 2-14
 transport user data, 2-14, 2-15



Computer Associates™