Concepts and Facilities

This chapter describes the basic functional concepts of the Cisco IOS for S/390 Application Program Interface (API) as well as the facilities used to send requests to the API. It includes these sections:

- API Organization

  Describes the relationship between the API, the transport service provider, and the transport service user. Also describes 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 of time.

- 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 of time.

- 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 Control

  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.

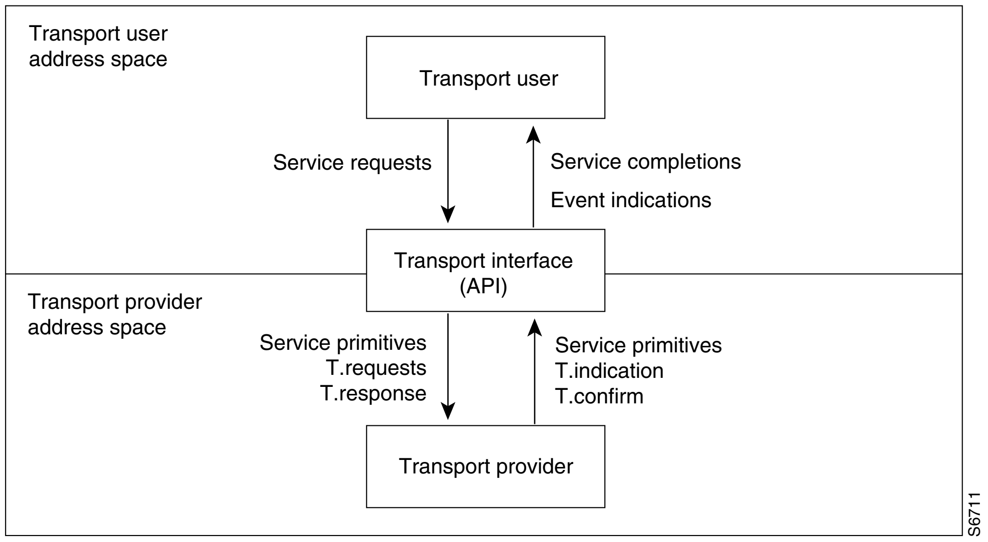
API Organization

The Cisco IOS for S/390 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 concern itself 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. Also, 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.

Figure 2-1 Transport User, Transport Provider, and API Relationship



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 has issued an indication or confirm primitive.

Service Request Processing

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

- Initial processing performed during invocation of the request
- Primary processing performed within the transport provider's address space
- 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

These modes are provided for processing service requests:

- Synchronous mode blocks the transport user until the request has been completed. In this case, the task issuing the request is suspended and is not redispatched (except to process asynchronous interrupts) until posted during the final processing phase.
- Asynchronous mode 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 event(s) occur, processing in the transport provider's address space is performed asynchronously, and the transport user is informed when the request is complete. 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 (for example, 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 begins with the transport provider and works up towards 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 and/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)

IFS is 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

Space-switched Program Call (PC) routines that are located in the transport provider's address space, but can be 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, and that generate the linkage to common storage interface routines to initiate service requests.

C Service Functions Library

A library of functions that provide services similar to the assembler language interface for programs written in the C programming language. 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.

Socket Interface Functions Library

A library of functions that implement a socket interface for application programs written in the C programming language. The API socket library enhances portability of networking applications developed to run on BSD UNIX systems or systems supporting the BSD socket interface.

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 OSI terminology introduced in Chapter <3>, Overview of API.

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

With connection-mode service, a connection is established for the purpose of transferring data, and all data is delivered intact, uncorrupted, in the same order as transmitted and without duplication.

Connectionless-Mode Service

With connectionless-mode, 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 COTS: a long-term binding between two CLTS users for the purpose of 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 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 unbinding 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 has been bound, are ready for sending and receiving data. COTS endpoints cannot engage in data transfer until connected.

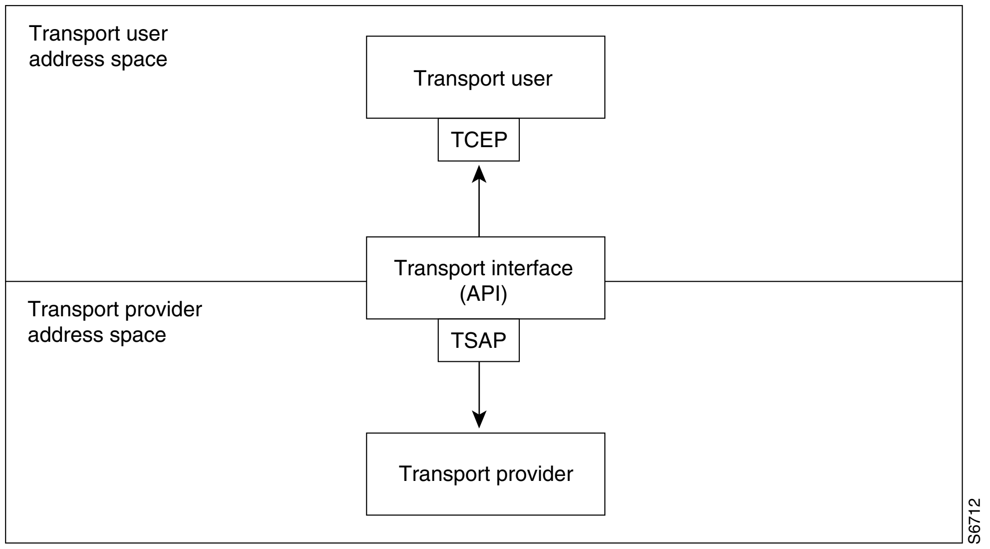
One of two connection strategies can be employed by the transport user. The determination of which to use is based on the role of the transport user and its relationship with other transport users with whom it connects. Generally one user is the provider of some service, and the other is a consumer.

The transport user providing a service (the server) operates in server mode. The user of a service (the client) 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 has been 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. This diagram illustrates client mode with one endpoint per TSAP:

Figure 2-2 Client Mode with One Endpoint per TSAP


Transport user address space
Transport user
TCEP
Transport interface (API)
Transport provider address space
TSAP
Transport provider
S6712

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 (in other words, establish) the connection or reject (in other words, abandon) the connection. If the connection is accepted, the endpoint enters the data transfer phase and a confirmation is returned to the client.

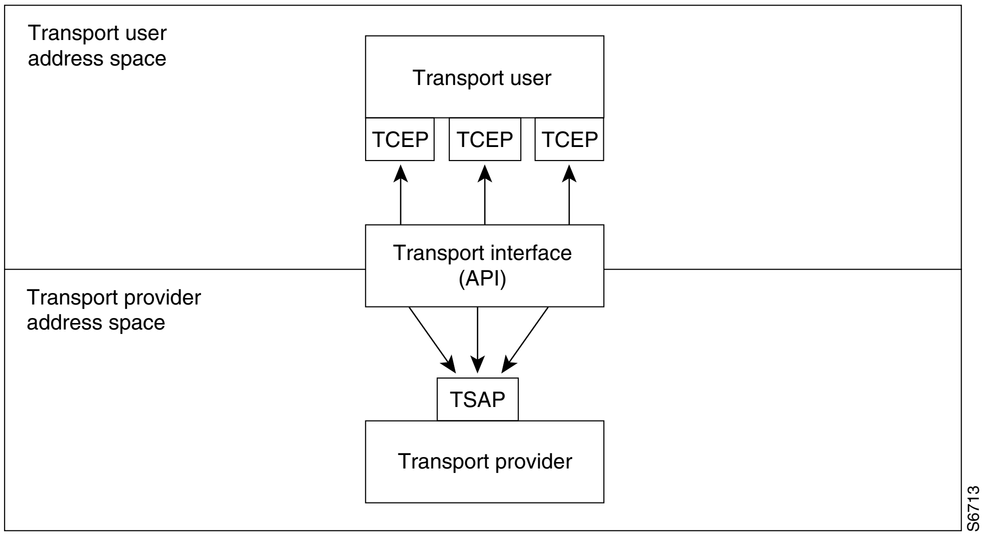
Single-threaded Servers

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 illustrates the client-server relationship in a single-threaded server.

Multi-threaded Servers

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

Figure 2-3 Multi-threaded Server with Multiple Endpoints per TSAP



Transport user address space
Transport user
TCEP
TCEP
TCEP
Transport interface (API)
Transport provider address space
TSAP
Transport provider
S6713

Working in multi-threaded 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 multi-threaded 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, 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

Transport Layer Interface (TLI) mode is the default data transfer mode and is used in all previous releases of Cisco IOS for S/390. TLI mode is a programmable interface that lets applications be built as 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 has been 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 and/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 of the data cannot be sent, none of it will be sent and the request will complete in error.

Socket Mode

Socket mode data transfer is a new feature of Version 5.2 of Cisco IOS for S/390. 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 upon 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 or the data may be sent. The amount of data that is actually sent depends upon the space available in the current send buffer.

Disconnect and Orderly Release

When two connected transport users have completed 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, and the method commonly supported by all transport providers, is the disconnect service. 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

The alternate method is to use an orderly release service 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*)*, and 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, etc.

The TPL is the primary structure for requesting API services and exchanging information. Read 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, and/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 (in other words, 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. A more thorough discussion of service parameters can be found in the *Cisco IOS for S/390 Assembler API Macro Reference*, which describes the operands of each macro instruction in detail.

Common Parameters

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

This is a list of these parameters:

- A TPL identifier
- The function code
- A semaphore indicating whether or not 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 which 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 might be 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 has been, 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. Read 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 (in other words, 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:

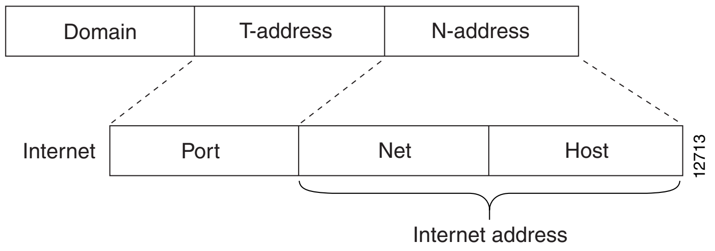
- Transport protocol address
- Transport user data
- Transport protocol options

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

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.

Figure 2-4 Transport Protocol Address 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 may be found in Appendix D, Data Structures of the *Assembler API Macro Reference*. The equivalent C structure declarations may be found in Chapter 2, C Language Structures, of the *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 a total of 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 have been made based on the destination address. A partial protocol address is specified by indicating the absence of the network address, either by its length or its contents set to zero.

Transport User Data

Transport user data is referred to simply as *user data* throughout the remainder of this manual. 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 (in other words, 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 an integral number of octets (eight-bit bytes). Usually, it is the presentation layer that 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.

Figure 2-5 Direct User Data Parameters

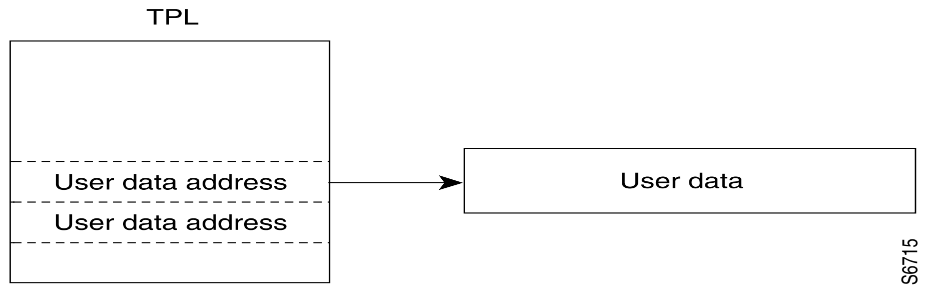
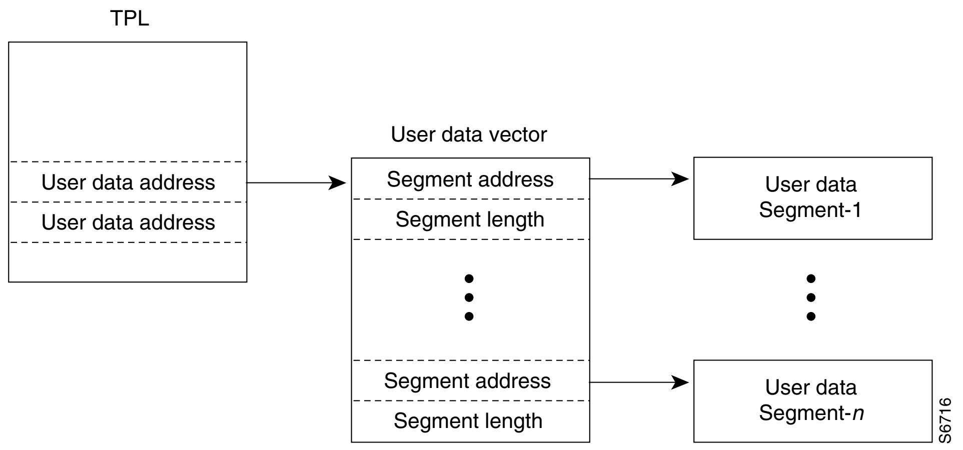


Figure 2-6 **Indirect User Data Parameters**



Transport Protocol Options

Transport protocol options are used for these reasons:

- To enable optional facilities
- To specify certain service parameters
- To 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 and/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 *Cisco IOS for S/390 Assembler API Macro Reference*.

The TPL is organized with the most frequently-used information at the beginning. 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. Read the *Cisco IOS for S/390 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

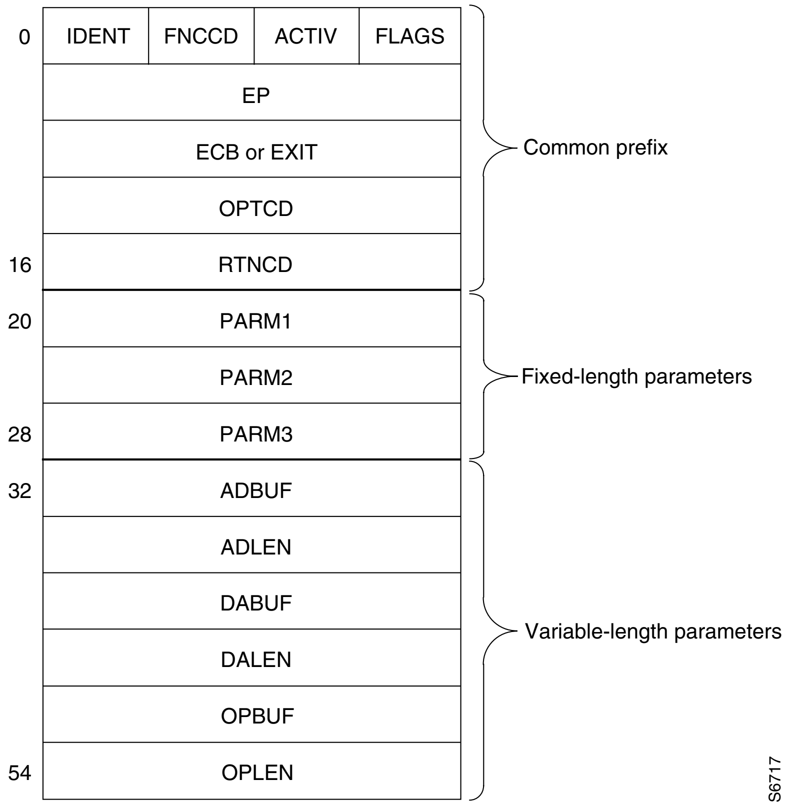
Option codes indicated with the OPTCD operand are stored in the OPTCD field. The symbol defined by the TPL dsect that corresponds to this field is constructed by prepending TPL to the name of the field. Therefore, the option codes are stored at the symbolic location TPLOPTCD.

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

TPL Standard Format

This diagram shows the standard format of a TPL containing a common, fixed-length, and variable-length parameter section:

Figure 2-7 Standard TPL Format



Common Prefix

The TPL begins 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, and it contains the minimum amount of information required to execute a transport service function.

This list describes the major fields within this prefix:

IDENT

This field 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. 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 or not 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. 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

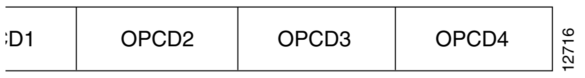
This field is used for synchronization and is shared by these different uses:

- 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 is composed of four consecutive bytes.

Figure 2-8 OPTCD Field Format



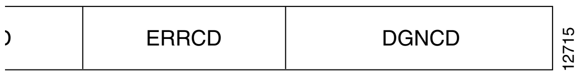
- OPCD1 specifies options that apply to all transport service functions
- OPCD2, OPCD3, and OPCD4 specify options that apply to specific functions or groups of functions

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

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

Figure 2-9 RTNCD Subfields



- ACTCD is a recovery action code that can be used to determine the appropriate action on completion of a transport service function.
- ERRCD is 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 table lists the fixed-length parameter fields:

PARM1

This field is 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

This field is 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.

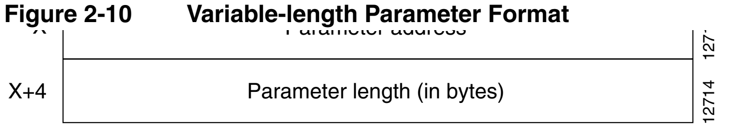
PARM3

This field is 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.

Figure 2-10 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, and 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 that is 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 2 byte module identifier and a 2 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. 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, 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:

Table 2-1 Session-level Services

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).

The 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 Application Program Control Block *(*APCB*)*. 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. It is the contents of the APCB at the time it is opened that 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 be used 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

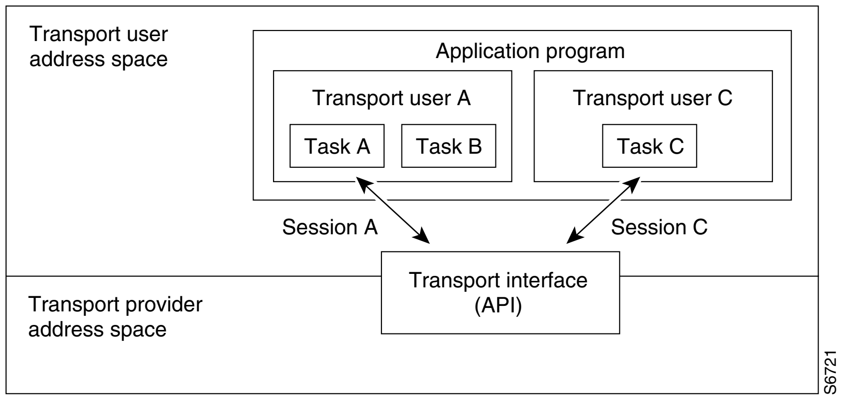
Until now this manual has referred to the application program and the transport user as if they were the same. Often they are, but there is a formal definition of the transport user that can now be stated:

- The term *application program* is used throughout the remainder of this manual to refer, in general, to the program running in an address space that is using the API.
- The term *transport user* is used specifically to refer 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 illustrates an application program that consists of tasks A, B, and C:

Figure 2-11 Application Program Example



Transport user A (represented by task A) and transport user C (represented by task C) has each opened an APCB and has 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. Also, an arbitrary word of user context can be specified 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.

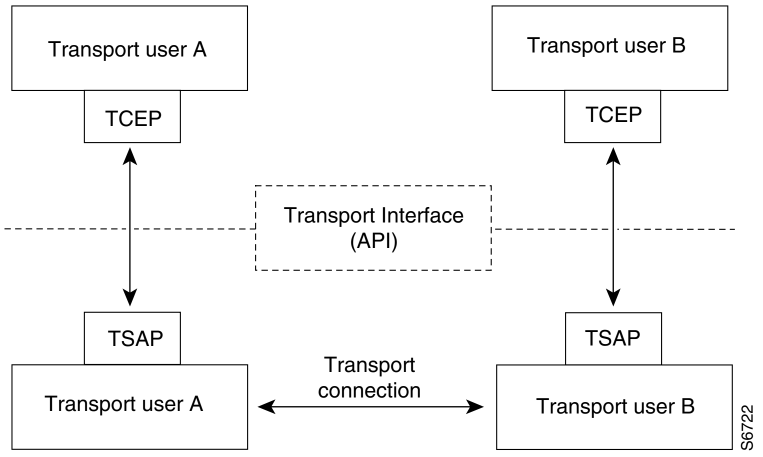
Connection-mode Service

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

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

This diagram illustrates this transport connection:

Figure 2-12 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 could 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

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

Table 2-2 Local Endpoint Management Functions:

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

In this table, and the tables 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.

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 have 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. 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

This 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
*                                   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
         .
        . [new endpoint ID can be used in subsequent requests]
         .
        TDSECT TPL                  GENERATE TPL DSECT
```

- 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 has already been stored in the TPL, perhaps by a previous macro instruction, it is not necessary to code this operand. 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 (for example, 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, TEXLST and TEVNTLST, are provided to generate an event list.

These protocol events are defined and correspond to particular service primitives issued by the transport provider:

Table 2-3 Protocol Events

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. Read 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 may be requested using Event Control Blocks (ECBs) rather than an exit. An ECB is an MVS control block which 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 is completed, 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.

A new macro, TEVNTLST, has been added to support protocol event notification ECBs. TEVNTLST supports ECBs and protocol event exits; TEXLST supports exits only. TOPEN uses a new parameter, EVENTLST, to support event lists consisting of exits and ECBs. EVENTLST is not supported on the APCB macro.

Fast-Path Authorized Exits

An authorized user may 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.

Example

If a protocol address has been 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

This 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 has been made to pass control of an endpoint from one task to another as long as each has opened an APCB. This capability has also been extended 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 has rendezvoused 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 has been 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 has been 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

An option code (OPTCD=USE) 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 example 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 (1) is recommended (for example, a single-threaded server), usually a value of five (5) 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, any address from a pool of available addresses can be assigned. 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. In other words, an address may 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 has been 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 may use the listen queue to prioritize incoming connection requests. Since connect indications do not need to be accepted in the order they were presented, the server may 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 has been 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     TUBNDERR           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 remote protocol address is the address of the connected peer transport user.

The local protocol address can be retrieved at any time after a successful TBIND has been issued and is requested by indicating OPTCD=LOCAL with the TADDR function. 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 dubious value since most of the information returned can be acquired through some other means. However, the local protocol address returned by TADDR after a connection has been 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 – Obtaining 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 run-time 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 Transport Service Information Block (TIB*)*. The TIB is mapped by a dsect generated by the TDSECT macro instruction and is listed in its entirety in *Cisco IOS for S/390 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 *Cisco IOS for S/390 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 (in other words, the OPTCD parameter) provided with the request indicates the action to be taken by the transport provider.

These services are available:

Table 2-4 TOPTION Services

DEFAULT	Return the default values of the indicated options
QUERY	Return the current values of the indicated options
VERIFY	Verify whether the value indicated for each option is supported, and if not, return the superior value supported
DECLARE	Set the specified options to the indicated values and return any options negotiated to an inferior value

Option codes (OPTCD) also indicate whether the options to be manipulated are provider options (TP) or interface options (API). Transport 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,OPLEN=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 may 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 in this way:

Figure 2-13 Protocol Options List Format

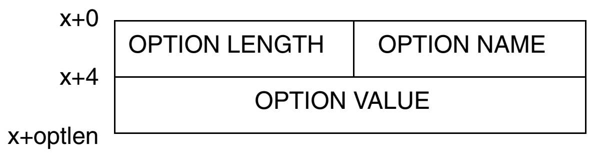


Table 2-5 Protocol Option Entries

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 (for example, 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 may 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 has connected, and even after data has been 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, 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 is authenticated with the resident security system, and access privileges associated with the user ID is 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 may 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.

This table lists the service functions in alphabetical order:

Table 2-6 API Service Functions

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
*                                   INITIATE CONNECT
          LTR   15,15               CONNECTION INITIATED?
          BNZ   TCONNERR            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)
*                                     FTP SERVER 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
```

The client must wait for confirmation. The client receives confirmation by issuing the TCONFIRM function. TCONFIRM does not complete until a connect confirm has been 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 has been received. In the former case, and when operating in synchronous mode, the client is suspended until the server has responded. In the latter case, an asynchronous exit routine is entered, giving notification that the confirmation has arrived. The TCONFIRM function should then be executed; it completes immediately without suspending the issuing task. Use of exit routines is described separately in 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 Multi-threaded 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.
- Multi-threaded 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 or not 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
*                                INITIATE LISTEN
          LTR    15,15           CONNECT INDICATION RECEIVED?
          BNZ    TLSTNERR        IF NOT, GO TO ERROR ROUTINE
          USING  TPL,1
          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 will cause the session to be abortively disconnected. Any data that was sent (from the client in this case), 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 may 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 run-time 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]
           .
```

Multi-threaded Servers

A multi-threaded 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
*                                   OPEN ENDPOINT
          LTR   15,15               ENDPOINT CREATED?
          BNZ   TOPENERR            IF NOT, GO TO ERROR ROUTINE
          USING TPL,1
          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)
*                                   ACCEPT TO NEW ENDPOINT
          LTR   15,15               CONNECTION ESTABLISHED?
          BNZ   TACPTERR            IF NOT, GO TO ERROR ROUTINE
           .
          . [the new endpoint is now ready for data transfer]
           .
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
```

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 multi-threaded mode and single-threaded mode are similar, except that multi-threaded 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 of time. The strategies that may 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 has been 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   TRETERR            IF NOT, GO TO ERROR ROUTINE
          USING TPL,1
          CLI   TPLERRCD,TENOLSTN  LISTEN ALREADY COMPLETED?
          BNE   TRETERR            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.

COTS Data Transfer Functions

This table summarizes the API service functions implemented for data transfer:

Table 2-7 COTS Data Transfer Functions

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

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 run-time 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 may 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 re-assembly, 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 run-time by examining the TIB returned by the TINFO service. The maximum size of a TSDU may be large (possibly unlimited), and in particular, may 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 (in other words, 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 exits, 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 TRECV 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 TRECV 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, and the API sets the option codes prior to the completion of TRECV. 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 has been sent and also allows swapping of the address space. Similarly, TRECV 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 may 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
         TRECV EP=(9),DABUF=REPLY,DALEN=L'REPLY RECEIVE REPLY
         LTR    15,15               REPLY RECEIVED?
         BNZ    TRECVERR            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 TRECV 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, then the request remains pending and is completed 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 TRECV 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 TRECV function has been 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.

This table summarizes the API service functions for connection release:

Table 2-8 Connection Release Functions

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 has not been 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.

```
***********************************************************************
*  TRECV 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 may also be used during connection establishment. It has already been mentioned that TREJECT is a special case of TDISCONN, issued by the server to reject a connection request. TDISCONN may 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 has been 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 TRECV function after all data has been received, or 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.

```
**********************************************************************
*  TRECV COMPLETED WITH ERROR -- CHECK FOR ORDERLY RELEASE
**********************************************************************
          USING TPL,1
TRECVERR  CH    15,=AL2(TRFAILED)   ROUTINE FAILURE?
          BNE   FATAL               IF NOT, NO RECOVERY
          CH    0,=AL2(TAINTEG)     DATA INTEGRITY ERROR?
          BNE   NOTRLSE             IF NOT, CAN'T BE RELEASE
          CLI   TPLERRCD,TERELESE   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 have been invoked by each transport user. The order in which they are invoked depends on which transport user initiates the release. Both users may initiate the release at the same time, and all race conditions are resolved by the API and the transport provider. TRELACK may also be issued before the indication is received. In this case, the request remains pending and is not completed 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. This mode is useful when data transfer is uni-directional, and the transport user does not expect to receive any data.

Once TRELEASE has been invoked, TSEND requests cannot be issued. Similarly, no TRECV request may be issued once TRELACK has been 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 (in other words, 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 that are 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 effect 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 may 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 just two phases: local endpoint management and data transfer. The local management phase is identical to connection-mode service in all but a few respects.

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 may 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 may 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 which, 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:

Table 2-9 Supported Asynchronous Protocol Exits

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. 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 may 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 may be used to retrieve local and remote protocol addresses. 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 may 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

This table lists the service functions supporting connectionless data transfer:

Table 2-10 CLTS Data Transfer Functions

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.

TSENDTO – 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, and 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  TRECVFR 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 (in other words, 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.

TRECVFR – Receiving Incoming Datagrams

The TRECVFR 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 also be identified. When the TRECVFR function completes, the data and source protocol address are returned in the storage areas provided, and their lengths are updated to reflect the actual amount of data returned.

A single datagram may 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 TRECVFR function and know that it completes immediately without suspending the issuing task for an extended period of time.

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 and TRECVFR function completes, and 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 with the TOPTION service.

A TSENDTO request may be completed as soon as the datagram has been passed to the transport provider. Therefore, subsequent protocol errors such as nonreachable destinations must be signaled to the transport user at some later time. If the transport user has enabled the datagram error exit, its exit routine is scheduled when a datagram error indication arrives. Otherwise, the next TSENDTO or TRECVFR request is completed with an error. In either case, the transport user must receive the indication by invoking the TRECVERR 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. Application programs that intend to remain protocol-independent should not interpret this error code and should use it for diagnostic purposes only.

```
*************************************************************************
*  TRECVFR 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(TAINTEG)         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 TRECVERR
           TRECVERR EP=(9),ADBUF=DGERRPA,ADLEN=L'DGERRPA
*                                        GET ERROR INFO
           LTR   15,15                   TRECVERR FAILED?
           BNZ   FATAL                   WHOOPS, HOW CAN THAT BE?
           L     2,TPLDGERR              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 may be engaged in a conversation with a server for an extended period of time. 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 of time. The association is established by the local transport user in exactly the same manner as a real transport connection would be 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 has been 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 connection-less 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 has been opened, local endpoint management proceeds as if a transport connection was 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, and 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
*                               OPEN ENDPOINT
        LTR     15,15           ENDPOINT CREATED?
        BNZ     TOPENERR        IF NOT, GO TO ERROR ROUTINE
        L       9,TPLEPID       LOAD NEW ENDPOINT ID
        TBIND EP=(9),ADBUF=SERVERPA,ADLEN=LTPAINET,QLSTN=1,           +
                OPTCD=USE       BIND SERVER TRANSPORT ADDRESS
        LTR     15,15           BIND SUCCESSFUL?
        BNZ     TBINDERR        IF NOT, GO TO ERROR ROUTINE
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 be associated 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
*                               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
          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. Also, 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 multi-threaded. 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 multi-threaded mode. When the association has been established, incoming datagrams are routed to the appropriate endpoint and are received by invoking the TRECV 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 TRECV 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 has been 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 used.

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

Table 2-11 Local Endpoint Control Functions:

Function	Parameters	M/O	Description
TCHECK	Endpoint ID	M	Wait for completion and schedule error recovery routine if completed abnormally.
TERROR	Endpoint ID	M	Analyze error and generate message in WTO list format.
TEXEC	Endpoint ID	M	Re-execute previous function.
TSTATE	Endpoint ID	M	Get 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, re-executing a previous request, and determining the state of an endpoint after the completion of a service function.

TCHECK

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:

- It synchronizes the application program with the completion of the service request
- It sets the TPL inactive (and reusable for another request)
- It schedules the appropriate error recovery routine if the request completed 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 Program Synchronization and Control.

```
***************************************************************************
*  LISTEN FOR CONNECTION REQUESTS IN ASYNCHRONOUS MODE
***************************************************************************
LISTEN    TLISTEN EP=(9),ADBUF=CLIENTPA,ADLEN=L'CLIENTPA,ECB=INTERNAL,   +
                OPTCD=ASYNC         LISTEN 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 generate 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 has been fully initialized, including the function code associated with a given service request. TEXEC is typically used in an error recovery routine to re-execute a previous request, or when the application program wants to bypass the API macro instructions and manipulate the TPL directly. 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 that are used by other macro instructions.

These are the declarative macro instructions supported by the API:

Table 2-12 Supported Declarative Macro Instructions

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

The APCB Macro Instruction

The APCB macro instruction generates the Application Program Control Block (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 (in other words, 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.

These are the defined data structures:

Table 2-13 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 TEXLST Macro Instruction

The TEXLST 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, which also defines the maximum length of the exit list as well as 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 TEXLST macro instruction defines the entry point of an exit routine that may be scheduled by the API to process certain asynchronous events. Read the *Cisco IOS for S/390 Assembler API Macro Reference* for more information on the TEXLST macro instruction.

Here is an example of this macro instruction:

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

The TEVNTLST Macro Instruction

The TEVNTLST macro instruction is used to define ECBs to post for protocol event notification. TEVNTLST supports protocol event exits as well. TEVNTLST follows the same rule as TEXLST, except there is no TOPEN, AOPEN, APEND, SYNAD or LERAD parameter. Unlike TEXLST, which may be referenced on TOPEN or AOPEN, only TOPEN may reference TEVNTLST. 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=(CONNX,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 either with 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.

A second 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 these factors:

- 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 will fail.

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.

This table defines these states and how they are related to the service modes for which they are valid:

Table 2-14 Endpoint States

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 has been opened, but no local protocol address has been bound to the endpoint.
2	TSDSABLD	COTS CLTS	Disabled A local protocol address has been bound to the endpoint, and the endpoint is disabled for queueing incoming connection requests (QLSTN=0). If the service mode is connectionless, the endpoint is ready for data transfer.
3	TSENABLD	COTS (CLTS,ASSOC)	Enabled A local protocol address has been bound to the endpoint, and the endpoint is enabled for queueing incoming connection requests (QLSTN>0).
4	TSINCONN	COTS (CLTS,ASSOC)	Connect-indication-pending One or more connect indications have been received which have not been accepted or rejected.
5	TSOUCONN	COTS (CLTS,ASSOC)	Connect-in-progress An outgoing connection is in progress, and the endpoint is awaiting connect confirmation.
6	TSCONNCT	COTS (CLTS,ASSOC)	Connected A connection (or association) has been established and the endpoint is ready for data transfer.

Table 2-14 Endpoint States (Continued)

State Value	State Name	Service Type	Description
7	TSINRLSE	(COTS,ORDREL) (CLTS,ASSOC)	Release-indication-pending An orderly release has been received and acknowledged, and the endpoint may continue sending data.
8	TSOURLSE	(COTS,ORDREL) (CLTS,ASSOC)	Release-in-progress An orderly release has been 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. Appendix A - “Endpoint State Transitions” lists all possible state transitions caused by any API service function.

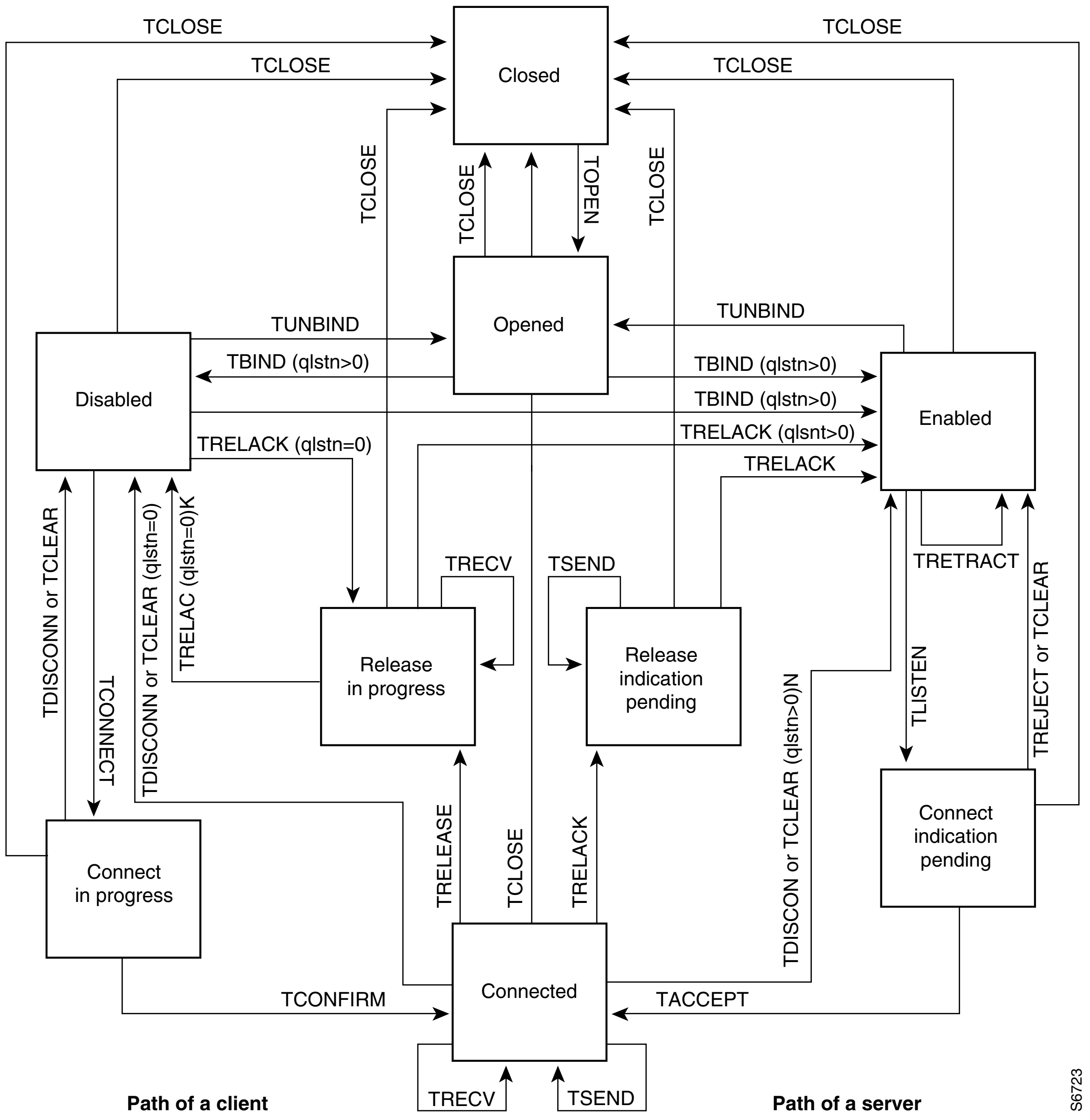
The following diagrams summarize the state transitions for connection-mode and connectionless-mode service.

Note Due to the many variations possible, only the common transitions are shown. Use Appendix A 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.

Connect-Mode States

This diagram illustrates the state transitions for connection-mode service. The left side of the diagram represents the client path and the right side represents the server.

Figure 2-14 States Transitions for Connection-mode Services



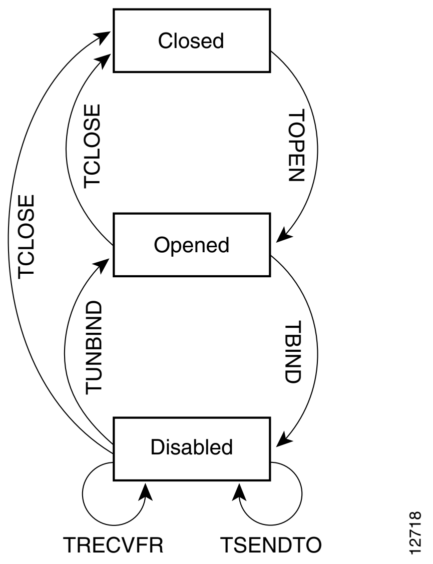
- An endpoint initially exists in the closed state and transits to the opened state when opened by the TOPEN service. The client may then bind a protocol address by executing a TBIND request
- A client should leave the endpoint disabled for receiving connect indications by specifying a QLSTN value of zero. This causes the endpoint to enter the disabled state.
- A server should specify a QLSTN value greater than zero when the protocol address is bound, and the endpoint transits to the enabled state. A TUNBIND function executed in the enabled or disabled state causes the endpoint to reenter the opened state.
- 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 TRECV 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. Also, TCLOSE can be issued at any time, returning the endpoint to the closed state.

Connectionless-Mode States

This diagram illustrates the state transitions for connectionless-mode service:

Figure 2-15 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. On completion of data transfer, the endpoint may be closed, returning it to the closed state, or the local protocol address may 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 be completed with an error. Also, 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 be overlapped or preempted by subsequent requests. This is a brief list of these exceptions:

- Multiple TSEND, TSENDTO, TRECV, and TRECVFR 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 TRECVs 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 has been completed. If multiple requests are to be 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 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. If this happens, 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 exhausts its resources before the API does.