Your Responsibility for Your Company’s Telecommunications Security

The final responsibility for securing both this system and its networked equipment rests with you - an Avaya customer’s system administrator, your telecommunications peers, and your managers. Base the fulfillment of your responsibility on acquired knowledge and resources from a variety of sources including but not limited to:

- Installation documents
- System administration documents
- Security documents
- Hardware-/software-based security tools
- Shared information between you and your peers
- Telecommunications security experts

To prevent intrusions to your telecommunications equipment, you and your peers should carefully program and configure:

- your Avaya-provided software applications, as well as their underlying hardware/software platforms and interfaces
- any other equipment networked to your Avaya products.

Avaya Fraud Intervention

If you suspect that you are being victimized by toll fraud and you need technical assistance or support, call Technical Service Center Toll Fraud Intervention Hotline at +1 800 643 2353 for the United States and Canada. For additional support telephone numbers, see the Avaya Web site: http://www.avaya.com

Select Support, then select Escalation Lists US and International. This Web site includes telephone numbers for escalation within the United States. For escalation telephone numbers outside the United States, select Global Escalation List.

Providing Telecommunications Security

Telecommunications security (of voice, data, and/or video communications) is the prevention of any type of intrusion to (that is, either unauthorized or malicious access to or use of) your company’s telecommunications equipment by some party. Your company’s “telecommunications equipment” includes both this Avaya product and any other voice/data/video equipment that could be accessed via this Avaya product (that is, “networked equipment”). An “outside party” is anyone who is not a corporate employee, agent, subcontractor, or working on your company’s behalf. Whereas, a “malicious party” is anyone (including someone who may be otherwise authorized) who accesses your telecommunications equipment with either malicious or mischievous intent.

Such intrusions may be either through synchronous (time-multiplexed and/or circuit-based) or asynchronous (character-, message-, or packet-based) equipment or interfaces for reasons of:

- Utilization (of capabilities special to the accessed equipment)
- Theft (such as, of intellectual property, financial assets, or toll-facility access)
- Eavesdropping (privacy invasions to humans)
- Mischief (troubling, but apparently innocuous, tampering)
- Harm (such as harmful tampering, data loss or alteration, regardless of motive or intent)

Be aware that there may be a risk of unauthorized intrusions associated with your system and/or its networked equipment. Also realize that, if such an intrusion should occur, it could result in a variety of losses to your company (including but not limited to: human/data privacy, intellectual property, material assets, financial resources, labor costs, and/or legal costs).
Contents

About This Document ix
- External Specification of the TSDI for Telephony Services ix
- Reason For Reissue ix
  Impact of Dial Plan Expansion x
- Scope of this Document x
- What’s In This Document x
- How to Use this Document xii
  Related Documents xii

1 Telephony Services Overview 1-1
- The Telephony Server and API Standards 1-1
- A Simple Configuration 1-1
- Terminology 1-3
- Basic Connectivity 1-4
- Telephony Server Architecture 1-4
- Telephony Server Software Modules 1-6
- The Telephony Services Driver Interface 1-6

2 Telephony Server Communication Model 2-1
- Telephony Server — A Client-Server Model 2-1
  Routing Services Reverses the Client-Server Roles 2-1
  Differences Between the OA&M Model and the CSTA Model 2-2
- PBX Driver Tasks and Responsibilities 2-2
  Driver Initialization 2-2
  Message-Based Interface 2-3
Contents

- Driver Termination 2-3
- Traffic Measurements Utility 2-3
- Telephony Server Multiplexing 2-7
- CSTA Tserver Security 2-7
  - Client Access Security Levels 2-7
- Security Issues and Class of Service 2-8
  - Home Worktop Class of Service 2-8
  - Monitor Class of Service 2-8
  - Control Class of Service 2-9
  - Routing Class of Service 2-9
- Driver OA&M 2-9
  - Using the Tserver as a Transport 2-9
- Error Log Interface 2-10

3 Functional Description 3-1
- PBX Driver to Tserver Interface 3-1
  - Driver Registration 3-1
    - Registration Mechanism 3-1
    - Driver Registration Security Level 3-3
  - TSDI Version Control 3-3
  - Traffic Measurements Registration 3-3
  - Receiving Requests and Responses 3-4
  - Sending Requests and Responses 3-4
  - Traffic Measurements Reporting 3-5
  - Driver to Tserver Heartbeat Message 3-6
  - Unregistering for Traffic Reporting 3-6
  - Unregistering the Driver 3-6
  - Telephony Server Flow Control Of TSDI Messages 3-6
    - Telephony Server Flow Control Overview 3-6
    - Selective Driver Flow Control 3-7
    - TSDI High-Water Mark Flow Control 3-7
- PBX Driver to Client Interface 3-8
  - Advertising Driver Services 3-8
  - The Stream to the Client Workstation 3-9

Issue 1 — December 2002
Contents

- TSAPI Version Control 3-42
- Private Data Definition 3-44
  - Private Data Version Negotiation 3-44
- Error Log Interface 3-46

4 TSDI Coding 4-1
- Initializing the Driver with the Tserver 4-1
  - main() 4-1
  - unloadDriver() 4-2
  - sanity() 4-3
- Traffic Measurement Functions 4-3
  - Traffic Registration 4-3
  - Traffic Unregistration 4-4
  - Maintaining Traffic State 4-5
  - Traffic Call Back Routine 4-6
  - Reporting Traffic 4-7
- Processing an acsOpenStream() Request 4-9
  - Receiving an ACSOpenStreamConfEvent 4-9
  - Returning an ACSOpenStreamConfEvent 4-10
  - Returning an ACSUniversalFailureConfEvent 4-11
- Processing an acsCloseStream() Request 4-12
- Creating a CSTAConferenceCallConfEvent 4-12
- Private Data 4-15
- Processing a Monitor Request 4-15

5 TSDI Manual Pages 5-1
- tdiAllocBuffer() 5-1
  - Parameters 5-1
- tdiDriverRegister( ) 5-4
- tdiDriverSanity( ) 5-8
- tdiDriverUnregister ( ) 5-9
Contents

- tdiFreeBuffer() 5-10
- tdiGetSessionIDInfo() 5-12
- tdiGetSessionInformation() 5-14
- tdiGetTDISize() 5-16
- tdiGetTservVersion() 5-17
- tdiLogError() 5-19
- tdiMapInvokeID() 5-21
- tdiMemAllocSize() 5-22
- tdiQueueSize() 5-25
- tdiReceiveFromTserver() 5-27
- tdiSendToTserver() 5-29
- tdiSetMessageFlowControl() 5-31
- tdiGetMessageFlowControl() 5-34
- tdiTrafficRegister() 5-36
- tdiTrafficReport() 5-39
- tdiTrafficUnregister() 5-41
- TSDI and CSTA Message Interface Header Files 5-44

6 OA&M Manual Pages 6-1
- Introduction 6-1
- srvDriverRequest() 6-2
- TSRVDriverOAMConfEvent 6-4
- TSRVDriverOAMEvent 6-5
- OA&M Header Files 6-7

7 Implementation Notes 7-1
- Overview 7-1
- Driver Requirements 7-2
  - tdiStartDriver() 7-2
  - tdiStopDriver() 7-3
- tdiDriverUnload 7-4

Issue 1 — December 2002
Contents

- Thread Priorities 7-5
- Compiling and Linking 7-5
- Porting from NetWare to NT 7-5
- Platform Differences 7-5
  - Threads 7-6
  - Memory allocation 7-6
  - Semaphores 7-6
- Server Console Interfaces 7-6
  - Porting existing NetWare Server console screens 7-6
- Tips on porting… 7-7
  - For the Driver DLL: 7-7
  - For the GUI application: 7-7
- Coding Examples 7-8
  - DllMain() 7-8
  - tdiStartDriver() 7-8
  - tdiStopDriver() 7-9

A  Issue/Release History  A-1
About This Document

External Specification of the TSDI for Telephony Services

This document is the external specification of the Telephony Services Driver Interface for the Telephony Services product. The Telephony Server provides service for Computer-Supported Telecommunications Applications (CSTA) and consists of three parts:

- **Application Programming Interface (API)**
  The API supported by the Telephony Server is based on the European Computer Manufacturers Association (ECMA) CSTA standard. The Telephony Server software supporting the API exchanges messages that represent the API function calls and parameters over the network with the Tserver.

- **Tserver**
  The Tserver is responsible for routing messages between the application workstations on the network and the PBX Driver.

- **PBX Driver**
  The PBX Driver implements the CSTA services via a switch vendor-specific Computer Telephony Integration (CTI) link to a PBX. The Telephony Services Driver Interface is an open interface that specifies how messages are passed between the Tserver and any vendor’s Driver.

Reason For Reissue

This document is reissued to accommodate technical changes driven by the seven digit DEFINITY Dial Plan expansion, which expands the maximum extension length on the dial plan from five to seven digits.
Impact of Dial Plan Expansion

When Avaya Computer Telephony cannot determine if a 7-digit extension is internal, it will be reported as an IMPLICIT_PUBLIC number. Any extension less than 7 digits will be designated as IMPLICIT_PRIVATE or EXPLICIT_PRIVATE_LOCAL_NUMBER (see "Creating a CSTAConferenceCallConfEvent" on page 4-12).

Scope of this Document

This document specifically covers the function call interface to pass the messages between the Ts erver and Driver, the structure of these messages, and requirements for vendors that must be followed when writing a PBX Driver that adheres to the interface (Chapter 7 of this document describes what a driver must implement for a Windows NT platform).

What’s In This Document

This section summarizes the contents of this document.

- The Preface, About this Document, provides you with an orientation to the topics covered in this document. It also provides a list of related documents that form the prerequisites for this document.
- Chapter 1, Telephony Services Architecture, describes how the Telephony Services Driver Interface relates to the Telephony Server and the Computer-Supported Telecommunications Applications (CSTA) API, defines terminology used throughout this document, and includes a list of documents that are prerequisites for this document.
- Chapter 2, Telephony Server Communications Model, describes the communication between the major components of the Telephony Server and provides a high-level description of the tasks and responsibilities of the PBX Driver, and describes services provided by the Telephony Server for the PBX Driver.
- Chapter 3, Functional Description, describes the Telephony Services Driver Interface from the PBX Driver perspective.
- Chapter 4, TSDI Coding, provides coding examples that can be used as the basic, skeletal outline of a driver which will be using the TSDI to process CSTA messages.
- Chapter 5, TSDI Manual Pages, contains the manual pages for all of the TSDI APIs. It also contains a list of TSDI and CSTA Message Interface Header Files.
- Chapter 6, OA&M Manual Pages, contains the manual pages for the OA&M APIs.
Chapter 7, Implementation Notes, provides operating system-specific implementation details and notes on compiling and linking the Driver.

Appendix A, Release History, contains a record of the changes to this document.
How to Use this Document

Since the Telephony Services Driver Interface is based on messages that represent the CSTA-API function calls and parameters, this document should be read in conjunction with the CSTA API specification [TSAPI].

Related Documents

This document assumes that the reader is familiar with the documents listed below. The first two documents are provided on the Avaya Computer Telephony CD-ROM.

Telephony Services Overview

The Telephony Server and API Standards

The Telephony Server provides the desktop integration of telephones and personal computers by exporting an API (application programming interface) over a network. The basis for the Telephony Services API is the Computer Supported Telecommunications Applications (CSTA) standard. The CSTA standard is based on the European Computer Manufacturers Association (ECMA) CSTA standard.

A Simple Configuration

Figure 1-1 provides a high-level view of a simple Telephony Server configuration. Each phone and associated computer represents a workstation on a desktop; workstations can run applications that are integrated with the local phone. Although Figure 1-1 shows one link to one switch, the Telephony Server (actually, the PBX driver) can have multiple links, possibly terminating on different switches.
The application uses the CSTA API supported by a native library loaded on the workstation.

The library supporting the CSTA API exchanges messages that represent API function calls and parameters over the network with the T server.

The T server is a module that resides on the Telephony Server and is responsible for routing messages between the applications on workstations connected to the network and the PBX Driver.

The PBX Driver resides on the Telephony Server and implements the CSTA services via a switch vendor-specific Computer Telephony Integration (CTI) link to a PBX.

A CTI link is a logical link between the computing environment (Telephony Server) and the switching environment (PBX). Each Driver that supports CTI links via the TSDI must define the physical implementation for a CTI link (that is, the CTI link could be one or more physical links).

The TSDI is an open interface that specifies how messages are passed between the T server and any vendor's Driver.

Figure 1-1. Schematic of a Telephony Services Implementation

- The application uses the CSTA API supported by a native library loaded on the workstation.
- The library supporting the CSTA API exchanges messages that represent API function calls and parameters over the network with the T server.
- The T server is a module that resides on the Telephony Server and is responsible for routing messages between the applications on workstations connected to the network and the PBX Driver.
- The PBX Driver resides on the Telephony Server and implements the CSTA services via a switch vendor-specific Computer Telephony Integration (CTI) link to a PBX.
- A CTI link is a logical link between the computing environment (Telephony Server) and the switching environment (PBX). Each Driver that supports CTI links via the TSDI must define the physical implementation for a CTI link (that is, the CTI link could be one or more physical links).
- The TSDI is an open interface that specifies how messages are passed between the T server and any vendor's Driver.
Terminology

Use this section as a quick reference to the technical terms used in this document.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS</td>
<td>Application Control Services</td>
</tr>
<tr>
<td>API</td>
<td>Applications Programming Interface. The API specifies the access methods that a programmer can use to exercise functionality provided by a kernel or library. An example of an API for this product is the “C” language interface used to access CSTA capabilities supported by the Telephony Server.</td>
</tr>
<tr>
<td>CSTA</td>
<td>Computer-Supported Telecommunications Applications.</td>
</tr>
<tr>
<td>CTI</td>
<td>Computer Telephony Integration</td>
</tr>
<tr>
<td>OA&amp;M</td>
<td>Operations, Administration and Maintenance. A module that provides the maintenance and administration interface.</td>
</tr>
<tr>
<td>PBX Driver</td>
<td>A vendor-dependent software specific to switch control. The PBX Driver is the service provider portion of the Telephony Server.</td>
</tr>
<tr>
<td>PDU</td>
<td>Protocol Data Unit. A data object exchanged between the Telephony Server and the client application.</td>
</tr>
<tr>
<td>Tserver</td>
<td>The specific module that manages the routing of CSTA requests and responses between a client application and the appropriate PBX driver. The Tserver is part of the Telephony Server.</td>
</tr>
<tr>
<td>Telephony Server</td>
<td>The Telephony Server comprises various components (Tserver, PBX Driver, Cserver) that together support CSTA.</td>
</tr>
<tr>
<td>TSDI (TDI)</td>
<td>Telephony Services Driver Interface: A connection between a Driver and the Tserver using a function call interface to pass messages between modules.</td>
</tr>
</tbody>
</table>
Basic Connectivity

Figure 1-2 illustrates the connectivity of a simple Telephony-Services configuration.

![Figure 1-2. A Simple Telephony Services Configuration](image)

Telephony Server Architecture

This section provides a high-level summary of the Telephony Server Architecture. Refer to Figure 1-2 as you read through the following list of components.

- **The Telephony Server** — The Telephony Server provides a platform for the desktop integration of telephones and personal computers. Figure 1-2 shows a block diagram of the Telephony Server architecture.

- **PCs and Workstations** — The personal computers or workstations can run CSTA applications that are integrated with their users phones. The CSTA applications will use a library (the CSTA API) as a client communicating with a Telephony Server that has advertised CSTA services on the network.

- **The CTI Link** — The Telephony Server will transport client requests over a switch vendor-specific link to provide switch integration (this is often referred to as the Computer Telephony Integration (CTI) link). Multiple applications may access Telephony Services on the same workstation simultaneously, and multiple workstations may request services from the same Telephony Server.
- **OA&M Services** — The Telephony Server provides a separate set of generic Operation Administration and Maintenance (OA&M) services. The OA&M application uses a separate native library (the OA&M API) as a client for communication with the Telephony Server. The OA&M API provides a basic interface for passing what appears to be a character array to the Telephony Server, between the OA&M application and the Driver.

---

**Format of OA&M Messages**

The format and interpretation of these messages are defined entirely by the OA&M application and the Driver. The Telephony Server supports one or more OA&M interfaces per PBX Driver (this is defined by the Driver).

---

![Telephony Server Architecture Diagram](image)

**Figure 1-3. Telephony Server Architecture**
Telephony Server Software Modules

The Telephony Server comprises three main modules (as shown in Figure 1-3):

- **Tserver**
  The Tserver manages Telephony and OA&M requests from clients (over the LAN). Additionally the Tserver performs the following functions.
  - confirms that each client is administered for the requested service, CSTA or OA&M
  - authenticates some CSTA (but not OA&M) requests, and
  - routes the requests to the appropriate PBX Driver.

- **Cserver**
  The Cserver manages client CSTA sessions on behalf of the PBX Driver and provides an ECMA-180 compliant interface to the PBX Driver. It is responsible for mapping CSTA requests and responses between the formats passed across the CSDI and TSDI interfaces. See [CSDI] for a full description of the Cserver.

- **PBX Driver**
  The PBX driver handles CSTA or OA&M requests for a specific vendor's PBX. The PBX Driver may choose to provide OA&M services through the Telephony Server, or the Driver may choose to provide its own OA&M interface.

The Telephony Services Driver Interface

The interface between the Tserver and the PBX Driver is a function call interface referred to in this document as the Telephony Services Driver Interface (or TSDI). The TSDI is used to pass messages that represent (CSTA or OA&M) requests and responses between the Client Application and the PBX Driver.

The Telephony Server provides services to clients distributed over the network. The Tserver handles service advertising, client authentication, connection setup, and connection tear down. Note that all CSTA client requests are multiplexed to a single stream per PBX driver registration; that is, each tdiDriverRegister() generates a different driver ID which the Driver then uses in tdiReceiveFromTserver() calls. All CSTA client requests that come over ACS streams which were opened up to the advertised name created on behalf of this registration are multiplexed to the single stream of messages that a driver receives by using tdiReceiveFromTserver(). A PBX driver may register for CSTA or OA&M services one or more times. Each CSTA registration corresponds to one logical CTI link supported by the driver.

Issue 1 — December 2002
Telephony Server Communication
Model

Telephony Server — A Client-Server Model

The Telephony Server to telephony application communication is based on a client-server model. The CSTA application behaves as a client requesting switching function and status reporting services (as defined in [ECMA-179]) from the Telephony Server via the CSTA API function calls defined in TSAPI. The Telephony Server will send responses for each client request, and the client application will receive each response as a confirmation event through the CSTA API (see TSAPI). The event reports generated by the Telephony Server as part of the event reporting services (see [ECMA-179]) are also received by the client application as events through the CSTA API.

Routing Services Reverses the Client-Server Roles

The client-server roles for the application and the Telephony Server are reversed for the computing function services (routing services) defined in [ECMA-179]. The Telephony Server will format computing function requests that the application will receive as events through the API, and the application will use CSTA API function calls to send the response back to the Telephony Server.

NOTE:
This document often refers to the Telephony Server-application relationship as a server-client relationship, even though these roles are reversed for the computing function services.
Differences Between the OA&M Model and the CSTA Model

The OA&M application communication model is similar to the CSTA model, except that the application always performs the role of the service requesting client. The PBX Driver defines the OA&M services it will support; the Telephony Server (and the OA&M API) only enforce the client-server messaging model as described in “CSTA Messaging Interface” on page 3-28.

PBX Driver Tasks and Responsibilities

The design of a PBX Driver for the Telephony Server begins when the vendor determines the CSTA and OA&M services the Driver will support. The CSTA switching function and status reporting services require the PBX driver to act as a server for incoming requests from the client application. The CSTA computing function (routing) services require the PBX Driver to act as a client, generating requests for the server application.

NOTE: The PBX Driver must fail any CSTA client request that the Driver does not support.

If the PBX Driver is supporting OA&M services through the Telephony Server, the set of supported maintenance services, and the OA&M messages that support requests and responses for those services must be defined by the PBX Driver authors. The following sections describe three tasks that the PBX Driver must implement to support these services. There may be other requirements placed on the Driver based on operating system implementation details (see Chapter 7, Implementation Notes.)

Driver Initialization

The PBX Driver must first register with the Tserver before CSTA or OA&M requests can be routed to the driver. When the PBX Driver registers with the Tserver, resources are allocated for the Telephony Services Driver Interface, and the Tserver will advertise on the network that the Driver is available to handle CSTA (CTI Link) or OA&M application requests. The PBX Driver must separately register for each CTI or OA&M Link the driver is going to support. These registrations result in completely separate interfaces, and message traffic across these interfaces is completely independent. When a PBX Driver registers with the Tserver for CSTA services it must also register with the Tserver for Traffic Measurements reporting.
Message-Based Interface

The PBX Driver must be able to handle incoming requests from the CSTA or OA&M application immediately after the driver has completed the registration process. The PBX Driver sends and receives messages that represent the CSTA or OA&M requests and responses to the application through a function call interface to the T server. The PBX Driver must always be prepared to negatively acknowledge client application requests that the Driver does not understand.

If the PBX Driver provides CSTA services, each time it sends a message to the PBX and receives a message from the PBX it must report this traffic statistic to the T server. This is done through one of the Traffic Measurements function call interfaces to the T server.

NOTE:
Nearly every message in the CSTA API TSAPI that is sent by a client to the Driver is a Request-Response type message. The Driver must always send a response message, either the defined positive acknowledgment (that is, cstaMakeCallConfEvent in response to a cstaMakeCall request) defined in the CSTA API or one of the two defined negative acknowledgment messages, acsUniversalFailureConfEvent or cstaUniversalFailureConfEvent, to every client request. The CSTA Routing messages flow in the opposite direction. Here the application acts as the routing server and the PBX takes the role of the client. (Routing is discussed in more detail in Chapter 3 of this document).

Driver Termination

The PBX Driver should always unregister with the T server before exiting (unloading), or the Driver can unregister with the T server any time it wants to stop handling CSTA or OA&M requests. The T server will halt the advertisement of the Driver services and free all resources associated with the Telephony Services Driver Interface when the PBX Driver unregisters. The PBX Driver cannot use the Telephony Services Driver Interface for the CTI or OA&M Link for which the register was originally done after the unregister operation has completed. The Driver can, however, re-register for CSTA or OA&M services. Prior to unregistering with the T server to stop handling CSTA requests, the PBX Driver that was registered for Traffic Measurements reporting must unregister for traffic reporting.

Traffic Measurements Utility

The T server’s Traffic Measurements Utility (TMU) allows customers to watch and tune their CTI configuration. The TMU measures the flow of traffic from a PBX to its clients and from these clients to the PBX. This utility is intended for use with PBX Drivers providing CSTA services. Measurements are collected throughout the system at various locations; PBX Driver (providing CSTA services), TSDI for
each Driver and the Tserver. The PBX Driver measurements are taken at its interface to the PBX. The PBX Driver measurements consist of the messages it sends to the PBX and the messages it receives from the PBX. The TSDI measurements are taken at its interface between the PBX Driver and the Tserver. The Tserver measurements are taken at its interface to the client.

Each location is responsible for reporting traffic information to the TMU for both uplink and downlink messages. The terms uplink and downlink refer to the direction of the message in relation to the PBX; downlink means that the message is being sent to the PBX from the client and uplink means that the message is being sent away from the PBX or to the client. The TMU will calculate and report the following measurements for the Driver each hour based on the traffic information provided by the Driver.

The following list describes what measurements the TMU will report for the PBX Driver based on the information it receives from the Driver. A later table will show the information that the Driver needs to provide the TMU to generate these measurements.

- At the uplink, PBX Driver interface (from the PBX), the TMU will report:
  - Message Arrival Rate
    Number of messages received from the PBX in the past hour.
  - Peak Message Arrival Rate
    Maximum message arrival rate detected over a series of discrete, non-overlapping time intervals. This time interval defaults to 15 seconds and can be administered through the Tserver’s TMU interface.

- At the downlink, PBX Driver interface (to the PBX), the TMU will report:
  - Message Departure Rate
    Number of messages sent to the PBX for the past hour..

- For the PBX Driver interface, the TMU will report:
  - Message Rejection Rate
    Number of messages rejected or denied service due to a lack of resources. These messages are not included in any other rate. For example, the PBX Driver receives a message and cannot allocate a buffer to properly service this request or must reject the request due to some form of flow control. The Driver would not report that it received a message but instead would report that it rejected a message.

The TMU will report the same type of information for the Tserver but in the opposite directions (for example, the message arrival rate is downlink for the Tserver, meaning the messages are arriving from the client.) The TMU will also report this information for the TSDI in both the uplink (messages sent from Driver to the Tserver) and downlink (messages sent from the Tserver to the Driver).
directions. Queue lengths and queued times are calculated as well. Combining the measurements from all of the interfaces the end user will be able to watch their CTI configuration and make changes as necessary. The diagram below illustrates the measurements available to the end user.

Figure 2-1. PBX Driver Traffic Measurements

PBX Drivers will report their traffic information to the TMU for each CSTA service that it registers with the TServer through \texttt{tdiDriverRegister()}. TSDI traffic interface routines allow a Driver to register with the Traffic Measurements Utility, report traffic data, and unregister with the utility. The Tserver requires that a Driver register for traffic reporting before it will accept any traffic data. The manual pages for these new routines are provided in Chapter 4 of this document.
The table below shows the data that the Driver must report (through tdiTrafficReport()) and which measurements result from that data. The calculations are performed once an hour and then the rates start back at zero. An option is available at the Tserver’s TMU interface to allow the administrator to specify the interval over which the peak arrival rate is calculated. This option will be defaulted to 15 seconds but can be set between 5 and 15 seconds.

**NOTE:**
It is assumed that a Driver will provide a traffic report to the TMU each time it receives or sends a message rather than processing multiple messages before it calls the traffic report interface. Using the latter method of reporting will affect the accuracy of the Peak Arrival Rate measurement.

<table>
<thead>
<tr>
<th>Driver Reports for each registered CSTA Service:</th>
<th>Traffic Measurements Utility Calculates:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Received an uplink message (from the PBX) and have the necessary resources to process it</td>
<td>Message Arrival Rate&lt;br&gt;Peak Message Arrival Rate</td>
</tr>
<tr>
<td>Sent a message downlink (to the PBX)</td>
<td>Message Departure Rate</td>
</tr>
<tr>
<td>Cannot service a message due to a lack of resources</td>
<td>Message Rejection Rate</td>
</tr>
</tbody>
</table>

Administrative commands are also available that allow the user to enable/disable the TMU for the entire system or for a specific location (for example, a specific Driver, a specific TSDI or the Tserver). The Traffic Measurements feature must be enabled for any measurements to be performed. Similarly, if the feature is disabled, then no measurements will be taken regardless of whether any of the interfaces are still enabled.

**NOTE:**
This option means that the TMU must be able to communicate this enabling information to the affected reporting interface(s) so the interface will know to stop or start sending traffic data. This will be accomplished through the TSDI. Part of the registration process requires the Driver to specify a callback function that the TMU will call when the administration of the measurements feature has changed for the specific driver interface or the entire feature.

Other administration commands allow thresholds to be set for the peak arrival rate, and when reached, the TMU will place an entry in the Tserver’s Error Log and send an alarm to the Tserver’s OA&M application if the alarm notification feature is enabled for the Tserver. The TMU also provides a Report Generation facility and a Traffic Log giving the end user the capability of tracking these measurements.
Telephony Server Communication Model

Telephony Server Multiplexing

The telephony server does not provide multiplexing services for the Driver. An ACS stream is opened to a Driver (that is, the CTI Link name as generated by a Driver registration), not to a device. The application must know ahead of time to which CTI Link it wants to open a stream. When CSTA requests are received for a device, the request is received by the Tserver over an ACS stream that terminates at a CTI Link. The request is forwarded to the driver if and only if, the security database indicates the device in the request can be controlled by the user (login) that opened the ACS stream and that the CTI Link which the ACS stream terminates on can control the device. If the security database indicates that more than one CTI Link can be used to control a device, the Tserver does not select a CTI Link to receive the CSTA request. The Tserver always forwards requests to the CTI Link where the ACS stream terminates.

The above description also applies to CSTA monitor requests. If an application or several applications send requests to monitor the same device to the same CTI Link, the driver must be able to handle the multiple monitors. The Tserver will not multiplex these monitors to a single request for the driver.

In the security database, users (logins) are given permission to control devices and all the CTI Links that can support (control) a device are indicated.

CSTA Tserver Security

The Tserver will provide telephony-based security services for drivers registering for CSTA functionality. Providing the security services within the Tserver allows applications to present a uniform platform across multiple vendor’s drivers. A driver that wants or needs to provide its own security mechanism can override the telephony-based security services, to a point, by indicating so in the tdiDriverRegister() routine. A user wishing to open an ACS stream will always require at least a Login on the file server on which the Tserver is loaded. The lowest level of security a driver can register with is TDI_NO_SECURITY which means the Tserver will not validate any message including an acsOpenStream() call; that is, a client may open an ACS stream without a valid login/password.

Client Access Security Levels

The Telephony Server supports a Security Database that allows the administration of users, groups of users, worktops, devices, lists of devices, and lists of CTI Links. Users are given names, unique login IDs, passwords, pointers to their worktop records, security level permission to monitor and query, control and route for some set of devices by specifying a group, and permission to perform OA&M functions on Tservers and Drivers registered with Tservers. Worktop records include a worktop name, LAN address of a PC, a pointer to a
device record for the primary telephone, and a pointer to a list of devices if more than one device (that is, telephone, fax, etc.) is associated with a worktop. Group records specify a list of devices that belong to that group. Device records contain the type of device (PC, phone, etc.), a device ID, a security level permission, location of the device, and a list of T servers/PBXs that support this device.

Security Issues and Class of Service

The Telephony Server defines four classes of service that can be granted to a user in the security database for the Telephony Server.

Home Worktop Class of Service

This class of service gives a user the privilege to control the device monitor or query the device or set of devices associated directly with a user's home worktop location and possibly the device or set of devices associated with a worktop location a user may login from which is not the user's home worktop location. A user's home worktop is defined as the worktop associated with the user's record in the Tserver SDB.

The home worktop class of service is based on two assumptions. The first is based on the assumption that the LAN Address can be trusted. The second assumes that the LAN address is not secure and can easily be faked in a packet sent to the Telephony Server.

If the administration is set up to believe the LAN Address (by disabling the Tserver OA&M Restrict User Access to Home Worktop option), the following algorithm is used to determine if a user has a home worktop privilege over a device:

- When a user is logging in from his/her home workstation, he/she gets privilege over the set of devices defined by his/her worktop record. When a user logs in from a workstation other than his/her home workstation, he/she gets privilege over his/her home workstation set of devices as well as the set of devices associated with the workstation he/she logged in from.

If the administration is setup to not believe the LAN Address (by enabling the Tserver OA&M Restrict User Access to Home Worktop option), the following algorithm is used to determine if a user has privilege over a device:

- Regardless of which workstation a user is logging in from, he/she gets privilege over the set of devices defined by his/her worktop record.

Monitor Class of Service

Monitor class of service gives the user privilege to perform monitoring functions on a set of devices specified by the administration of this user. Three types of monitoring privileges can be granted to a user:

Issue 1 — December 2002
Device monitoring on a device: A list of devices is defined on which this user can perform device monitoring.

Call monitoring on a device: A list of devices is defined on which this user can perform call monitoring.

Call monitoring on a call: A user can be given permission to call monitor calls. Because calls are not known in advance, a specific list of calls that might be monitored cannot be administered. The user either has or does not have permission to do this type of monitoring.

Control Class of Service

Control class of service gives the user privilege to perform control functions only. A user must have some type of monitoring class of service to monitor devices.

Routing Class of Service

Routing class of service gives the user privilege to perform routing functions on a set of devices.

Driver OA&M

Using the Tserver as a Transport

The PBX Driver vendors can implement a client or server based application using the generic Telephony Server Driver OA&M API for driver OA&M (see Chapter 6 of this document). The Tserver would act as the transport mechanism in this case. To use the Tserver transport mechanism and OA&M API, the PBX Driver must register with the Tserver for OA&M functionality. The driver would then use the TSDI routines to pass vendor-defined OA&M messages analogous to CSTA messages for CSTA functionality. The Tserver will treat an OA&M message as a block of data received from a client, and the message will be passed directly to the PBX Driver that has registered for the OA&M services. The data contained within the message block is to be defined by the PBX Driver authors and is specific to each vendor's driver.

NOTE:
The generic OA&M messages that the Tserver will transport for use in defining a driver-specific OA&M application are simple character array (whose length is defined by the Driver and OA&M application). The Driver and OA&M application must be very careful in defining structures that will be overlaid on this byte array due to different byte ordering and padding rules employed on different client and server machines.
The T server will provide administration options on User Records that will provide access to the OA&M link for specific users. That is, each user in the T server Security Database can be given permission to open a stream to the Driver OA&M service (defined by the Driver registering for OA&M services).

The PBX Driver vendor may also select to provide OA&M services directly from the Driver module, from a separate Driver OA&M module, or through a client-based application where the Driver provides the transport mechanism. Regardless of how the PBX Driver vendor supports/supplies Driver OA&M services, the Telephony Server OA&M client application must still be used to administer and maintain the Telephony Server.

**Error Log Interface**

A common error log serves both the T server and the PBX driver. The interface to the error log is through the Telephony Server OA&M client application or the T server's Console Administration screens. It supports a standard function call interface so errors have a uniform appearance in the error log. The error log interface provides six severity levels: TRACE, CAUTION, AUDIT_TRAIL, WARNING, ERROR, and FATAL for errors, and will include the date, time, location of the error, a specific error code, and supporting text for each error (see “Error Log Interface” in “Error Log Interface” on page 2-10). Whenever possible, the PBX Driver should log errors in this common error log with the appropriate severity. The PBX Driver authors are free to ignore this error log interface and provide their own mechanism, but their error messages will not be integrated with the Telephony Server.
Functional Description

PBX Driver to Tserver Interface

The TSDI provides a function call interface with a mechanism for the exchange of messages representing CSTA requests and responses that map to the CSTA API (see TSAPI), or OA&M requests and responses that map to the OA&M API (see “OA&M Interface” on page 3-41. The PBX Driver must first register with the Tserver, and then the driver can send and receive these requests and responses using Telephony Services Driver Interface routines. All messages exchanged with the Tserver must be allocated via the tdiAllocBuffer() routine (see “tdiAllocBuffer()” on page 5-1); they cannot be directly allocated from the operating system. The PBX Driver must always unregister with the Tserver before it unloads.

The following sections provide a brief description of the function call interface provided by the Telephony Services Driver Interface routines. See Chapter 5 for a complete specification of the interface routines.

Driver Registration

Registration Mechanism

A PBX Driver must establish a connection with the Tserver before it can provide the CSTA services described in the CSTA API TSAPI, or OA&M services described in the <Link> “OA&M Interface” section, on page 58. A separate connection must be created for each type of service the PBX Driver will provide (for example, one for CSTA services and one for OA&M services). Each CTI Link or OAM Link supported by the Driver for which the Driver wants the Tserver to advertise this service must be registered separately with the Tserver; that is, each TSDI registration supports one CTI or OAM link.

➤ NOTE:

A CTI Link is a logical link connecting the Driver to the PBX. The CTI Link can be one or more physical links. A Driver should register separately for each CTI Link that it will support.
The Driver must use the Telephony Services Driver Interface routine, \texttt{tdiDriverRegister()}, to establish a connection between the Driver and the T server. When a PBX Driver registers via the \texttt{tdiDriverRegister()} routine, the T server creates a separate TSDI instance by allocating resources from the OS for this (TSDI), and begin service advertising on behalf of the PBX Driver. The Telephony Server treats each \texttt{tdiDriverRegister()}, or TSDI instance completely independently. From the Telephony Servers viewpoint each \texttt{tdiDriverRegister()} represents a different Driver, even though multiple registrations may have been done by the same Driver.

The T server will apply the \texttt{vendor\_name}, \texttt{service\_name}, and \texttt{service\_type} parameters provided by the Driver in the \texttt{tdiDriverRegister()} routine to advertise this service (CSTA or OA&M) to clients. This routine will guarantee that no two Driver registrations result in the same advertised name. The T server will also store the \texttt{driver\_name} parameter provided by the PBX Driver in the \texttt{tdiDriverRegister()} routine for maintenance purposes. Version information must be specified in the \texttt{tdiDriverRegister()} routine so that the PBX Driver can guard against version compatibility problems with the T server. Because two versions of the TSDI currently exists, the version field must always be set to \texttt{TSDI\_VERSION} or \texttt{TSDI\_VERSION\_2} (see “TSDI Version Control” on page 3-3).

The Telephony Services Driver Interface monitors TSDI buffer usage by both the PBX Driver and the T server. The PBX Driver can specify a maximum number of bytes that can be allocated (by the PBX Driver and the T server) for TSDI messages via the \texttt{buffer\_descriptor} parameter of the \texttt{tdiDriverRegister()} routine. The T server will allocate space from the TSDI interface only for buffers that are used to forward/send events to the driver. All other space allocated by the T server is not charged against the total TSDI space.

The \texttt{tdiDriverRegister()} routine returns a \texttt{driverID} to the PBX Driver that must be used to identify this TSDI connection. All message buffer allocations, send requests, receive requests, and unregister requests for this TSDI connection (registration) must use this \texttt{driverID}. The PBX Driver is not allowed to interchange messages from one TSDI registration to another. The \texttt{tdiDriverRegister()} routine is a blocking function that will return to the PBX Driver after the T server has initiated the service advertising procedures.
Driver Registration Security Level

For a Driver to be certified, it must register with the `TDI_CSTA_SECURITY` option. The valid security options a Driver can register with are defined below:

- **TDI_CSTA_SECURITY**
  - Login and Password will be validated on the `acsOpenStream()` request.
  - Entry in the T server's Security Database must contain this login. This is also checked at the time of the `acsOpenStream()` request.
  - Each subsequent CSTA request will be validated per the user's administered permissions.

- **TDI_LOGIN_SECURITY**
  - Login and Password will be validated on the `acsOpenStream()` request.
  - Entry in the T server's Security Database must contain this login. This is also checked at the time of the `acsOpenStream()` request.

- **TDI_NO_SECURITY**
  - No validation is done on an `acsOpenStream()` request.

TSDI Version Control

The `version` field in the `tdiDriverRegister()` function is used to enforce version control of the TSDI. Currently, two versions, `TSDI_VERSION` and `TSDI_VERSION_2`, of the TSDI exist and the `version` field must be set to one of these.

Version 2 of the TSDI allows the PBX Driver to specify the `TSAPI` versions that the PBX Driver supports in the `protocol_descriptor` argument to `tdiDriverRegister()`. This argument replaces the previously unused `channel_number` argument. See “TSAPI Version Control” on page 3-42 for more information on `TSAPI` version control.

Version 2 of the TSDI also adds the new function `tdiGetTservVersion()` that allows a driver to determine the customer and internal version of the T server and the supported `TSAPI` protocol versions for a given service type. This function is not available in the Release 1 T server. If the PBX Driver is being written to work with both the Release 1 and Release 2 T servers, then this function should be imported dynamically.

Traffic Measurements Registration

Once a PBX Driver registers with the T server to provide CSTA services (not OAM), it must register with the Traffic Measurements Utility of the T server. The Driver will do this by using the routine, `tdiTrafficRegister()`. This utility as a whole measures the traffic from a client to a PBX and from this PBX to a client.
This enables end users to watch and fine-tune their CTI configuration. Specifically for the Driver, this utility will measure the message arrival rate from a PBX and the message transmission rate to the PBX. Given the uplink arrival rate (from the PBX) reported by the Driver, the Traffic Measurements Utility will calculate the peak message arrival rate over a series of discrete non-overlapping time intervals. The PBX Driver may also report that it had to reject a message, or in other words, it could not allocate the necessary resources to successfully process the request. The Traffic Measurements Utility will keep track of this rejection rate as well.

The traffic registration interface routine requires the driverID be returned to the PBX Driver by the tdiDriverRegister() routine and is not blocking.

Receiving Requests and Responses

The tdiReceiveFromTserver() routine is used by the PBX Driver to receive incoming (CSTA or OA&M) requests and responses from the Client. These Client requests and responses are contained in message buffers (see “The Message Format Between the PBX Driver and the Client” on page 3-10) returned by the bufptr parameter to the tdiReceiveFromTserver() routine. The tdiReceiveFromTserver() routine is a blocking routine that will only return when a message buffer is ready for the PBX Driver, or an error has occurred. The PBX Driver “owns” the message buffer returned by the tdiReceiveFromTserver() routine. The message buffer should not be directly returned to the OS by the PBX Driver instead it must be returned back to the Telephony Services Driver Interface instance. The buffer can be returned by either calling tdiFreeBuffer() or populating it with a request or response message and sending it back across the TSDI to the client (see the next section, “Sending Requests and Responses”) via tdiSendToTserver(). The tdiFreeBuffer() routine is a non-blocking routine which requires the driverID and a pointer to the buffer to be released. The same driverID that was used in the tdiReceiveFromTserver() routine must be applied to the tdiFreeBuffer() routine.

Sending Requests and Responses

The tdiSendToTserver() routine is used by the PBX Driver to send outgoing (CSTA or OA&M) requests and responses to a Client. These Client requests and responses are contained in message buffers (see “The Message Format Between the PBX Driver and the Client” on page 3-10) pointed to by the bufptr parameter of the tdiSendToTserver() routine. The message buffers must be "owned" by the PBX Driver, and they must be allocated from the Telephony Services Driver Interface that will be used to send the messages to the Client. (The driverID returned from the tdiDriverRegister() routine must be used to allocate the message buffer and send the message buffer.) Message buffers are "owned" by the PBX Driver if the Driver has received the message buffer from the Tserver (via
the tdiReceiveFromTserver() routine as described in the "Receiving Requests and Responses" section, on page 20), or allocated the message buffer from the TSDI via the tdiAllocBuffer() routine.

The tdiAllocBuffer() routine will return a BYTE aligned block of data as big as that requested by the PBX Driver, or the routine will return a failure indication. The tdiAllocBuffer() routine will fail the "request" if the size of the message buffer requested exceeds the maximum buffer size allowed in the OS, or if the size requested plus the size of all message buffers currently allocated by the PBX Driver and the Tserver (on this Telephony Services Driver interface) exceeds the limit specified by the Driver during PBX Driver registration. The TSDI memory allocation for this driver (or TSDI registration) will be charged the size of the data block allocated.

The tdiSendToTserver() routine supports a two-level message priority scheme. The PBX Driver can send "priority" messages through the interface by setting the priority parameter to TDI_PRIORITY_MESSAGE. The Telephony Services Driver Interface will always deliver priority messages (in First-In-First-Out order) before delivering "normal" messages (also in First-In-First-Out order).

The PBX Driver no longer "owns" a message buffer that was successfully passed to the Tserver (and the Client) via tdiSendToTserver(), and the PBX Driver should no longer access this buffer. The tdiSendToTserver() routine is a non-blocking routine that will fail only when the bufptr, priority, or driverID parameters are invalid.

NOTE:
The tdiReceiveFromTserver() routine is blocking because it waits on a semaphore that is only signaled when the Tserver sends the Driver a message. The tdiSendToTserver() routine, however, is non-blocking because it signals a different semaphore that the Tserver waits on for receiving messages from the Driver.

Traffic Measurements Reporting

The PBX Driver must call the interface routine, tdiTrafficReport(), when it receives a message from the PBX and when it sends a message to the PBX. In the case where the Driver cannot process a message due to a resource problem (for example, cannot allocate memory or other necessary resources), then it should report this as a rejection and not as an arrival. Rejections are also reported using the tdiTrafficReport() routine.

NOTE:
It is assumed that a Driver will provide a traffic report to the Traffic Measurements Utility each time it receives or sends a message rather than processing multiple messages before it calls the traffic report interface. Using the latter method of reporting will affect the accuracy of the Peak Arrival Rate measurement.
This interface routine requires the *driverID* returned to the PBX Driver by the `tdiDriverRegister()` routine. This routine will not block.

### Driver to Tserver Heartbeat Message

The PBX Driver must inform the Tserver, once a minute, that it is still active by calling the `tdiDriverSanity()` function. This routine requires one parameter, the *driverID*, returned to the PBX Driver by the `tdiDriverRegister()` routine. This is a non-blocking routine. If the Driver fails to call this function, the Tserver will generate a high severity error message that by default is placed in the error log file and sent to the Tserver’s OA&M client. No other recovery action is taken.

### Unregistering for Traffic Reporting

The PBX Driver must unregister with the Traffic Measurements Utility before it unregisters with the Tserver to stop handling CSTA requests. The interface routine for unregistering with this Utility is `tdiTrafficUnregister()`. It requires the *driverID* returned to the PBX Driver by the `tdiDriverRegister()` routine and is not a blocking call.

### Unregistering the Driver

The PBX Driver must unregister before unloading, or any time it needs to break the TSDI connection. The `tdiDriverUnregister()` routine requires a single parameter, the *driverID*, returned to the PBX Driver by the `tdiDriverRegister()` routine. This routine will not block, but only signal the Tserver that the driver wishes to unregister. After the routine returns, the Tserver will then forward the remaining messages sent from the Driver to the Tserver via the `tdiSendToTserver()` to the appropriate clients, and clear all of its resources from this TSDI. No resources allocated for this interface should be accessed by the PBX Driver after the `tdiDriverUnregister()` routine completes successfully.

### Telephony Server Flow Control Of TSDI Messages

### Telephony Server Flow Control Overview

The TSDI provides two routines that the PBX Driver can access to monitor the message flow for a specific interface. The `tdiMemAllocSize()` routine provides the amount of memory (in bytes) allocated for message buffers on this interface by the PBX Driver and the Tserver, and the `tdiQueueSize()` routine provides the count of messages queued to the Tserver and to the PBX Driver for this TSDI. The *driverID* that identifies the interface is the only input to these routines.
The tdiQueueSize() and tdiMemAllocSize() routines return structures that describe the current state of the message queues and the bytes allocated for message buffers for both the Tserver and the PBX Driver. The PBX Driver can use this information to determine if some form of flow control is required for messages exchanged between the PBX Driver and the Client.

The Telephony Server provides two types of flow control mechanism to drivers. The first is controllable by the driver in which the driver uses the tdiSetMessageFlowControl() routine to specify which CSTA and/or ACS messages should be flow controlled and at what level of message congestion this flow control should take affect. The second flow control mechanism is activated automatically by the Tserver when the size of allocated memory in the TSDI hits the defined TSDI High-Water Mark level. At this level, the Tserver rejects all messages from clients.

Selective Driver Flow Control

If a driver wishes to have specific CSTA or ACS messages flow controlled by the Tserver before the TSDI High-Water Mark flow control mechanism kicks in (see the “TSDI High-Water Mark Flow Control” on page 3-7), the driver must call the tdiSetMessageFlowControl() routine. This function allows the driver to specify which messages should be flow controlled and what level of TSDI message congestion the flow control should be activated. The TSDI message level is defined to be the number of TSDI messages either queued for the Driver or allocated to the Driver. When the Tserver receives a client request, it allocates a TSDI buffer and then queues the buffer for the Driver via the tdiSendToDriver() routine. When the driver receives the message via the tdiReceiveFromTserver() routine, that TSDI buffer is now considered to be allocated by the Driver. The Tserver will flow control messages sent to the Driver by rejecting those messages indicated in the tdiSetMessageFlowControl() routine with an error code of TSERVER_DRIVER_CONGESTION.

There is no flow control of messages sent from the driver to the Tserver as a result of the tdiSetMessageFlowControl() routine or as a result of any other action taken by the Tserver or Driver.

TSDI High-Water Mark Flow Control

The Tserver will flow control messages sent to the Driver via the tdiSendToDriver() routine by comparing the amount of TSDI resources currently allocated with the hiwater_mark. If it reaches this mark the Tserver will reject all new requests (including acsOpenStream()) with an error code of TSERVER.Driver_CONGESTION.

NOTE:

The hiwater_mark level specified by a driver in the original tdiDriverRegister() is stored in the Tserver Security Database. On subsequent loads of the driver, the values saved in the SDB are used and...
the buffer descriptor in the registration is ignored. The values can be change via the Tserver's OA&M application.

This will prevent new requests from being sent to the driver and should allow the driver to catch up on current requests. If the TSDI memory has hit the max_bytes level, then the Tserver will drop an ACS Stream when a new request comes in with an error code of TSERVER_NO_TDI_BUFFERS. If the Server is out of memory, then the Tserver will drop all requests for new ACS Streams with an error code of TSERVER_NO_MEMORY.

This means the driver should choose the max_bytes and hiwater_mark of the buffer_descriptor argument to the tdiDriverRegister() very carefully.

There is no flow control of messages sent from the driver to the Tserver.

### PBX Driver to Client Interface

#### Advertising Driver Services

The Tserver connection that was created when the PBX Driver registered via the tdiDriverRegister() routine is used for exchanging messages between the Driver and its clients. Clients using the CSTA API will attempt to create a CSTA stream to a PBX Driver that had previously registered for CSTA services. Clients using the OA&M API (see Chapter 6) will attempt to open an OA&M stream to a PBX Driver that had previously registered for OA&M services. The Tserver advertises the specific services registered by the PBX Driver via the tdiDriverRegister() routine. The Tserver uses the combination of the vendor_name, service_name and service_type parameters passed to tdiDriverRegister() in order to create the advertised name. The name will appear as follows:

<table>
<thead>
<tr>
<th>8 Bytes</th>
<th>14 Bytes</th>
<th>4 Bytes</th>
<th>&lt;= 19 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor_name</td>
<td>service_name</td>
<td>service_type</td>
<td>Server Name</td>
</tr>
</tbody>
</table>

In the following examples, assume that the name of the server on which the Tserver is running is BUFFALO

<table>
<thead>
<tr>
<th>Description</th>
<th>Example Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tserver OA&amp;M</td>
<td>TSERVER#TSRV_OAM#OAM#BUFFALO</td>
</tr>
<tr>
<td>Avaya CSTA Simulator</td>
<td>AVAYA#CSTASERV#CSTA#BUFFALO</td>
</tr>
</tbody>
</table>
The Stream to the Client Workstation

The stream between a client workstation and the PBX Driver can only be created after the Driver has registered with the T server for a specific service (CSTA or OAM).

Figure 3-1. CSTA and OA&M Streams
The CSTA client will perform an `acsOpenStream()` for the advertised name to establish a CSTA stream to the PBX Driver and set the `streamType` parameter to `ST_CSTA`. This `acsOpenStream()` request will be mapped to a corresponding CSTA message (see “The Message Format Between the PBX Driver and the Client” on page 3-10) and sent over the TSDI to the PBX Driver. If the Tserver does not reject the open request based on login and password, the request will be passed to the PBX Driver, which must then consider whether to honor this request or not. The Driver must send an ACS message back across the interface either acknowledging (`ACSOpenStreamConfEvent`) or failing (`ACSUniversalFailureConfEvent`) the open request. If the PBX Driver acknowledged the CSTA open request, a (logical) CSTA stream or session has been established between the PBX Driver and the client application. CSTA requests and responses that map to the CSTA API can be sent back and forth across the Telephony Services Driver Interface in the form of CSTA messages until the stream is disconnected by the PBX Driver (via the `ACSAbortStream`) or the client (via the `acsCloseStream()` or `acsAbortStream()` function calls).

The establishment of an OA&M stream is accomplished in the same way as the establishment of a CSTA stream as described above except the `streamType` parameter is set to `ST_OAM`.

A `Client Session ID` is included in each message exchanged on the stream between the PBX Driver and the CSTA or OA&M client. This `Client Session ID` uniquely identifies a session and is initially provided to the PBX Driver in the message corresponding to the `acsOpenStream()` performed by the client application. The PBX Driver must include this `Client Session ID` in each request or response sent across the TSDI for this client session. A Driver Control block is always included as the first portion of each message exchanged on the stream, and the `Client Session ID` is a mandatory field in the Driver Control block.

### The Message Format Between the PBX Driver and the Client

All messages exchanged between the PBX Driver and the client application (via the API) over the TSDI conform to a format that consists of a Driver Control block followed by (an optional) message block followed by (an optional) private data block. The Driver Control block is a fixed length structure starting at the first byte of the message buffer, and the Driver Control block has the same format for all messages exchanged over the TSDI.

The first two fields of the Driver Control block specify the location and length of the message block. The message block is a portion of the messages exchanged between the PBX Driver and the client over the TSDI. It is variable in length, and its format depends on the `messageClass` and `messageType` defined in the Driver Control block portion of the message.
The next two fields of the Driver Control block specify the location and length of the private data block. The private data block is an optional portion of the messages exchanged between the PBX Driver and the client over the TSDI. It is variable in length, and its format is defined beforehand between the client and the Driver.

The remaining Driver Control block fields are defined in the following sections.
The Driver Control Block

The first bytes of the messages exchanged across the TSDI must be the Driver Control block (DC block). The DC Block is the portion of the message passed across the TSDI to the PBX Driver that provides control information to the Driver for it to properly interpret and process client requests and responses. The PBX Driver must always create a DC block as the first part of request or response messages sent across the TSDI so that the Tserver can route the message to the appropriate client, and the client can receive the appropriate event. The DC block is populated by the Tserver with information received from the client. It is a C structure of fixed size that is located at the first byte of the message buffer that is passed across the TSDI and received by the PBX Driver via the tdiReceiveFromTserver() routine. The DC block must also start at the first byte of the message buffer sent to the Tserver via the tdiSendToTserver() routine. A message block and/or private data block that contains the formatted request or response message may immediately follow the DC block in the message buffer as defined by the message and private offset fields.

The DC block, the message block and private data block must be contained within one contiguous TSDI buffer that was either received by the driver via the tdiReceiveFromTserver() routine or allocated by the Driver via the tdiAllocBuffer() routine. There may be "holes" in the buffer between these three blocks as long as the DC block starts at the first byte and the message and private data offset fields are set correctly.

The Driver Control Block Field Definition

```c
/* Driver Control Block Structure */
typedef struct {
    unsigned short messageOffset;
    unsigned short messageLength;
    unsigned short privateOffset;
    unsigned short privateLength;
    InvokeID_t invokeID;
    CSTA_MonitorCrossRefID_t monitorCrossRefID;
    SessionID_t sessionID;
    EventClass_t messageClass;
    EventType_t messageType;
    short class_of_service;
} TDIDriverControlBlock_t;
```
messageOffset

The messageOffset is a value that determines the start of the request or response message associated with this Driver Control block. The messageOffset must be added to the address of the Driver Control block to get the start of the message. This field should be set to 0 if no message block is included with the Driver Control block.

messageLength

The messageLength is a value that provides the length of the request or response message associated with this Driver Control block. This field must be set to 0 if no message block is associated with this Driver Control block.

privateOffset

The privateOffset is a value that determines the start of the private data associated with this Driver Control block. The privateOffset must be added to the address of the Driver Control block to get to the beginning of the private data. This field should be set to 0 if no private data is included with the Driver Control block.

privateLength

The privateLength is a value that provides the length of the private data associated with this Driver Control block. This field must be set to 0 if no private data is associated with this Driver Control block.

invokeID

The invokeID is a value that is used for pairing request-response messages. The PBX Driver will receive an invokeID in each request message, and this value should be passed back unchanged in the DC block of any response message. The invokeID is undefined for requests or events that originated from the PBX Driver.

monitorCrossRefID

The monitorCrossRefID is a value for pairing monitor request-response messages. This monitorCrossRefID is set only in unsolicited events sent by the Driver in response to a previously opened monitor.

Note: The cstaMonitorStartConfEvent message includes a monitorCrossRefID in the actual structure and this is the place where a driver should indicate the cross-reference ID for that monitor, not here in the DC block.

sessionID

The sessionID is used to properly route messages to the appropriate client application. The PBX Driver will obtain this ID when it receives a message corresponding to an acsOpenStream() request from a client application, and this sessionID must be included in the Driver Control block for each message passed on the stream. The driver must populate this field of the DC block with the appropriate sessionID of the client whenever a confirmation report or unsolicited event for a client is sent via the tdiSendToTserver() routine.
**messageClass**  
The *messageClass* enumerates the message class for the message. The tables in the following sections list the possible values for *messageClass* depending on *messageType*. Message classes fall in three general categories: confirmation, solicited and unsolicited events. See the table below for a list of all possible message classes.

**messageType**  
The *messageType* indicates the type of message block following the DC block in the message buffer. The *messageType* field defines each message within a *messageClass*. See the “ACS Messaging Interface” on page 3-22 and “CSTA Messaging Interface” on page 3-28 for the *messageType* definitions for CSTA messages, and Chapter 6 for the *messageType* definitions for OA&M messages.

**class_of_service**  
The *class_of_service* field is not used in this release of the Tserver.
The following table lists all *messageClasses* that a driver can receive.

<table>
<thead>
<tr>
<th>MESSAGE CLASS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACSREQUEST</td>
<td>ACS request messages sent by a client to the driver (e.g., acsOpenStream, etc.). The <strong>API calls</strong> in Chapter 4 of [TSAPI] define the set of messages that make up this class.</td>
</tr>
<tr>
<td>ACSUNSOLICITED</td>
<td>ACS messages generated asynchronously by the Driver or Tserver to be sent to a client (e.g., ACSUniversalFailureEvent, etc.). See the structure <strong>ACSUnsolicitedEvent</strong> in <code>acs.h</code> for a <strong>complete</strong> enumeration of the messages that make up this class.</td>
</tr>
<tr>
<td>ACSCONFIRMATION</td>
<td>ACS confirmation messages sent by the Driver to a client in response to a previous ACS request message (e.g., ACSOpenStreamConfEvent, ACSUniversalFailureConfEvent, etc.). See the structure <strong>ACSConfirmationEvent</strong> in <code>acs.h</code> for a <strong>complete</strong> enumeration of the messages that make up this class.</td>
</tr>
<tr>
<td>CSTAREQUEST</td>
<td>CSTA request messages sent by a client to the driver (e.g., cstaMakeCall, etc.), or CSTA request messages sent by the driver to a client (e.g., CSTARouteRequestEvent, etc.). See the structure <strong>CSTAREquestEvent</strong> in <code>csta.h</code> for a <strong>partial</strong> enumeration of the messages that make up this class. The rest of the messages that make up this class are defined by the <strong>API calls</strong> in [TSAPI].</td>
</tr>
<tr>
<td>CSTAUNSOLICITED</td>
<td>CSTA messages generated asynchronously by the Driver to be sent to a client (e.g., CSTAEstablishedEvent, CSTADeliveredEvent, etc.). See the structure <strong>CSTAUnsolicitedEvent</strong> in <code>csta.h</code> for a <strong>partial</strong> enumeration of the messages that make up this class.</td>
</tr>
</tbody>
</table>
## MESSAGE CLASS DESCRIPTION

<table>
<thead>
<tr>
<th>MESSAGE CLASS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTACONFIRMATION</td>
<td>CSTA confirmation messages sent by the Driver to a client in response to a previous CSTA request message (e.g., CSTAMakeCallConfEvent, CSTAClearConnectionConfEvent, etc.) or a confirmation message sent by a client to a Driver in response to a previous CSTA request message (e.g., cstaRouteSelect, etc.).</td>
</tr>
<tr>
<td></td>
<td>See the structure CSTAConfirmationEvent in csta.h for a complete enumeration of the messages that make up this class.</td>
</tr>
<tr>
<td>CSTAEVENTREPORT</td>
<td>CSTA message sent by the Driver to a client that reports on some CSTA event, but which does not require the client to send a confirmation event in response.</td>
</tr>
<tr>
<td></td>
<td>See the structure CSTAEventReport in csta.h for a complete enumeration of the messages that make up this class.</td>
</tr>
<tr>
<td>TDRVRREQUEST</td>
<td>Driver OA&amp;M request messages sent by a client to the driver (e.g., TSRVDriverOAMReq, etc.).</td>
</tr>
<tr>
<td></td>
<td>The API calls in Chapter 6 of this document define the set of messages that make up this class.</td>
</tr>
<tr>
<td>TDRVRUNSOLICITED</td>
<td>Driver OA&amp;M messages generated asynchronously by the Driver to be sent to a client (e.g., TSRVDriverOAMEvent, etc.).</td>
</tr>
<tr>
<td></td>
<td>See the structure TSRVDriverUnsolicitedEvent in tdrv.h for a complete enumeration of the messages that make up this class.</td>
</tr>
<tr>
<td>TDRVRCONFIRMATION</td>
<td>Driver OA&amp;M confirmation messages sent by the Driver to a client in response to a previous Driver OA&amp;M request message (e.g., TSRVDriverOAMConfEvent, etc.).</td>
</tr>
<tr>
<td></td>
<td>See the structure TSRVDriverConfirmationEvent in tdrv.h for a complete enumeration of the messages that make up this class.</td>
</tr>
</tbody>
</table>
**DC Block / Message Class Mapping**

Only certain fields in the DC block are valid for each different message class defined in the table above for messages sent from the T server to the Driver or from the Driver to the T server (and then onto clients). The tables below depict which fields a Driver should expect to be set when it receives a message from the T server and which fields a Driver must fill in for messages it sends to the T server.

**Tserver to Driver Messages**

The tables below indicate with an X which fields are set by the T server when a message of this class is sent to the Driver. The Driver should only consider these fields to be valid for this class of message.

<table>
<thead>
<tr>
<th>DC Block Field</th>
<th>ACS REQUEST</th>
<th>CSTA REQUEST</th>
<th>CSTA CONFIRMATION</th>
<th>TDRVR REQUEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageOffset</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>messageLength</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>privateOffset</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>privateLength</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>invokeID</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>monitorCrossRefID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sessionID</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>messageClass</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>messageType</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>class_of_service</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. **Note**: The only message sent from the T server to the Driver of this class is a **CSTARouteSelect()** message.

The following class of messages are never sent from the T server to a Driver:

- ACSUNSOLICITED
- ACSCONFIRMATION
- CSTAUNSOLICITED
- CSTAEVENTREPORT
- TDRVRUNSOLICITED
- TDRVRCONFIRMATION
### Driver to Tserver Messages

The tables below indicate with an X which fields are set by the Driver when a message of this class is sent to the Tserver. The Tserver should only consider these fields to be valid for this class of message.

<table>
<thead>
<tr>
<th>DC Block Field</th>
<th>ACS CONFIRMATION</th>
<th>ACS UNSOLICITED</th>
<th>TDRVR CONFIRMATION</th>
<th>TDRVR UNSOLICITED</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageOffset</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>messageLength</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>privateOffset</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>privateLength</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>invokeID</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>monitorCrossRefID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sessionID</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>messageClass</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>messageType</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>class_of_service</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following class of messages are never sent from the Driver to a Tserver:
  - ACSREQUEST
  - TDRVREQUEST

The Protocol on the Client Stream

The three distinct phases for the communication between the PBX Driver and the client application include:
  - opening the stream (ACS messages),
  - the request-response protocol (CSTA or OA&M messages), and
  - closing the stream (ACS messages).

These phases are similar for both the CSTA and OA&M streams connecting the PBX Driver and the client application. Each phase consists of a set of messages that define the phase, and each message maps to either a (CSTA or OA&M) API function call issued by the client application, or a (CSTA or OA&M) event that will be presented to the client.
Details on how to handle the opening and closing of an ACS stream are given in the "ACS Messaging Interface" section, on page 39. Details on how to handle the request-response protocol for CSTA messages is given in the "CSTA Messaging Interface" section, on page 45 and details on handling the request-response protocol for Driver OA&M messages is given in the "OA&M Interface" section, on page 58.

TSDI Session ID to ACS Handle Mapping

The Session ID in the DC block is used by the T server to route Driver messages (confirmation or unsolicited) back to client applications. The Session ID is generated by the T server and represents a map between sessions and transport layer identifier for the T server. The ACS Handle that is used at the client API level to indicate which ACS Stream a message is to be sent out on or which stream a message was received on is created and managed totally by the client API library. The client API Library maintains a mapping between the ACS Handle the transportID (client side).

There is no direct mapping between TSDI Session IDs and ACS Handles. The Driver receives the Session ID in the DC block with the ACSOpenStream message when the stream is opened and will receive the Session ID in the DC block in all subsequent messages received over this stream. The Driver only needs to deal with Session IDs.

Scope of Monitor and Routing

Cross-Reference IDs

Monitor and Routing Cross-reference IDs generated by a Driver must be unique within the scope of a TSDI Driver Registration. A TSDI Driver Registration advertises a single logical CTI Link (the physical implementation of a logical CTI Link is defined by the Driver) and corresponds to the set of all ACS Streams opened to that CTI Link.

Scope of Invoke IDs

The Tserver guarantees that invokeIDs passed to the driver will be unique within an ACS stream. The Tserver actually saves the client generated invokeID (either application generated or library generated) for each request and creates a new, unique invokeID that is passed to the driver. The Tserver then tracks each confirmation event from a driver and replaces the invokeID in the DC block with the saved invoke ID before the event is sent back to the application. A driver can use the exported TSDI routine, tdiMapInvokeID(), to determine the actual client-generated invokeID. A driver should always use the client invokeID in any trace or debugging information it generates related to a stream.

NOTE:
The Tserver will discard invokeIDs after two minutes if no confirmation event is received. A driver has this time to send back a confirmation event to a request and still be guaranteed that the application will receive the
confirmation with the same invokeID that the application sent. If the Driver sends the confirmation after the invokeID times out, the Tserver will still forward the message to the client application but the invokeID will be the one generated by the Tserver, not the one sent by the application.

ACS Messaging Interface

This section describes the functions and events and their associated C structures of the Application Control Services (ACS) that are presented to the driver across the TSDI. ACS functions deal with the characteristics of the API interface (for example, opening and closing the CSTA or OA&M interface). They provide the ability to open, initialize, close, and manage a virtual communication channel (CSTA or OA&M stream) with any Telephony Server defined by the system. See TSAPI for more details on the ACS API C Language function calls and events.

ACS request, response, and event messages transported across the TSDI are presented in a C structure format and are used to establish and maintain ACS streams. Each API call and event that needs to be passed across the TSDI maps directly to a C structure defined in acsdefs.h. This C structure will be passed in the Message Block portion of the buffer where messageOffset points. A driver writer needs only to map the correct C structure, based on the messageClass and messageType, onto the Message Block to access the information contained in the ACS message.

Each of these categories has associated API calls and events that are illustrated below. The tables indicate the function name of the ACS API call or the ACS event name, its associated C structure name (a full C structure definition is included in acsdefs.h), the message class and type of the C structure, and the associated C structure name of the confirmation event.

The naming conventions for the items in the tables are as follows:

<table>
<thead>
<tr>
<th>acs&lt;function name&gt;()</th>
<th>acs API function call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS&lt;function name&gt;Event</td>
<td>ACS Event (request, confirmation or unsolicited)</td>
</tr>
<tr>
<td>&lt;function name&gt;_t</td>
<td>C structure name corresponding to an ACS API call, confirmation event or unsolicited event</td>
</tr>
<tr>
<td>&lt;define name&gt;</td>
<td>message type name</td>
</tr>
</tbody>
</table>
Application Control Services

Processing ACS Control Messages

Processing an acsOpenStream Message

For a new client session, the PBX driver will first receive the "open request" (CSTA or OA&M) message. The open request is sent to the driver so that the driver has an opportunity to accept or reject the open. This gives the driver control of how many streams it can handle and allows the driver to implement any type of resource high-water mark that may be appropriate. For example, a driver may wish to limit the amount of memory that is allocated based on the size of the server machine which could translate into limiting the number of open streams. Sending the open request to the driver also allows the driver to reject the open stream request if some error condition exists that would prevent CSTA requests from being processed (that is, the CTI Link is down).

NOTE:
The recommended way for a driver to handle the case where the CTI Link goes down is to keep the ACS streams open and send all client applications a system status message.

The "open request" message is created when the client performs an acsOpenStream() function call with streamType set to ST_CSTA for CSTA streams or ST_OAM for OA&M streams. The "open request" message is a block of memory passed across the TSDI from the Tserver to the Driver and contains...
the following: messageOffset, messageLength, messageClass, messageType, privateOffset, privateLength, invokeID, monitorCrossRefID, and a class_of_service. The sessionId identifies the (new) Client session or stream. The messageClass will identify the message as a CSTA, ACS or OA&M message. Each of these three classes is actually broken down into several classes. In the case of an acsOpenStream(), the messageClass will be ACSREQUEST.

The messageType defines which type of message is within the indicated messageClass. In the case of an acsOpenStream(), the messageType will by ACS_OPEN_STREAM. The messageType for all ACS, CSTA and OA&M messages is enumerated in the tables in the <Link> “ACS Messaging Interface” section, on page 39; the <Link> “CSTA Messaging Interface” section, on page 45; and the <Link> “OA&M Interface” section, on page 58, respectively. The messageClass and messageType indicate what type of message (that is, C-structure) is in the Driver Control block. The invokeID is used to implement the request-response protocol from the client application to the PBX Driver. The invokeID is generated by the (acsOpenStream()) API function call (it is also allowable for the application to specify the invokeID). This invokeID must be returned in the “open response” message so that the application can match it to a previous “open request”.

In Release 2 of Telephony Services standard mechanisms have been established for negotiating the TSAPI version and private data version, if applicable, to be used over a client stream. The TSAPI version control mechanism utilizes the apiVer field of the ACSOpenStream message and is described in the <Link> “TSAPI Version Control” section, on page 59. The private data version negotiation mechanism utilizes the private data associated with the ACSOpenStream message and is described in the <Link> “Private Data Definition” section, on page 61.

If the PBX Driver is intent on honoring the "open request", it must generate an ACSOpenStreamConfEvent message using the sessionId and the invokeID from the acsOpenStream() message. The PBX Driver must set the messageClass and messageType to the values defined for an ACSOpenStreamConfEvent, which are ACSCONFIRMATION and ACS_OPEN_STREAM_CONF, respectively. The PBX Driver must also set the apiVer, libVer, tsrvVer, and drvrVer fields within the ACSOpenStreamConfEvent message. All of these values, except the drvrVer, should be copied from the ACSOpenStream request message.

If the PBX Driver is not intent on honoring the "open request", it must generate an ACSUniversalFailureConfEvent message using the sessionId and the invokeID from the acsOpenStream() message. The PBX Driver must set the messageClass and messageType to the values defined for an ACSUniversalFailureConfEvent, which would be ACSCONFIRMATION and ACS_UNIVERSAL_FAILURE_CONF, respectively.
The "open response" message is sent back to the client through the API as a confirmation event. A successful exchange of the open request message and the positive open confirmation response message between the PBX Driver and the client application will result in the creation of an ACS stream (for either CSTA or OA&M services) between the PBX Driver and the client.

Processing a Close Request

The PBX Driver may receive a request from the client application to close the ACS stream (CSTA or OA&M). The "close request" message is created when the client application performs an `acsCloseStream()` function call (for CSTA streams), or an `acsAbortStream()` function call (the Tserver might also generate an `acsAbortStream()` message for the driver in certain failure conditions). The "close request" message contains the `sessionID`, the `messageClass`, the `messageType`, and possibly an `invokeID`. The `sessionID` identifies the client session or stream, and the `messageClass` will identify the message as a CSTA or OA&M message.

Processing an `acsCloseStream()`

In the case of an `acsCloseStream()`, the `messageClass` field will be `ACSREQUEST` and the `messageType` field will be `ACS_CLOSE_STREAM`. The `invokeID` is generated by the `acsCloseStream()` API function call. This `invokeID` must be returned in the "close response" message, `ACSCloseConfirmationEvent`, so that the application can match it to a previous "close request". The PBX Driver must generate an `ACSCloseConfirmationEvent` message in response to a `acsCloseStream()` using the `sessionID`, and the `invokeID` from the `acsCloseStream()` message, with `messageClass` and `messageType` set appropriately for an `ACSCloseConfirmationEvent` message; that is, `messageClass` set to `ACSCONFIRMATION` and `messageType` set to `ACS_CLOSE_STREAM_CONF`. Unlike the situation on an "open request" where the Driver can choose to accept or reject the open request, a Driver must always accept a "close stream" request and clean up any resources associated with that stream.

Processing an `acsAbortStream()`

In the case of an `acsAbortStream()`, the `messageClass` field will be `ACSREQUEST` and the `messageType` field will be `ACS_ABORT_STREAM`. There is no `invokeID` on this message since no reply from the Driver is required. The Driver must always accept an "abort stream" request and clean up any resources associated with that stream.
Asynchronously Closing a Stream from the Driver

When the Driver wishes to terminate an ACS Stream it must send an **ACSUniversalFailureEvent** with the error code set to **DRIVER_ACSHANDLE_TERMINATION**. When the T server sees this message (with this error code) from a Driver, the message will be delivered to the client and the ACS stream will be terminated.

When To Use
**ACSUniversalFailureConfEvent**

An **ACSUniversalFailureConfEvent** should always be used to negatively acknowledge an ACS request such as `acsOpenStream()`.

An **ACSUniversalFailureConfEvent** can also be used to NACK a CSTA or OA&M request such as `cstaMakeCall()`, **but only** if the reason for the NACK is related to allocating or maintaining resources related to the ACS Stream the `cstaMakeCall()` was received on. If the Driver is not going to positively acknowledge the CSTA or OA&M message with the defined Confirmation Event for that message, and the problem is not related to the ACS Stream, then the Driver should use a **CSTAUniversalFailureConfEvent** as defined by **TSAPI**, [ECMA/52] and [ECMA-179].

When To Use **ACSUniversalFailureEvent**

An **ACSUniversalFailureEvent** is an unsolicited, asynchronous message generated by the Driver (or T server) and sent to a client to indicate a failure of an existing ACS Stream.

Failure Codes To Use in **ACSUniversal**

Failure Type Messages

Both **ACSUniversalFailureConfEvent** and **ACSUniversalFailureEvent** share the same set of error codes, and the T server and Driver must also share this set of error codes for use in these messages. Error code values from 1-999 are reserved for the T server and values from 1000 and above are available for Driver use. Currently, a small number of generic Driver error codes are defined in the **ACSUniversalFailureEvent** section of [TSAPI].

The relevant portions of this chapter are reproduced below.

```c
typedef enum ACSUniversalFailure_t {
    :
    :
    DRIVER_DUPLICATE_ACSHANDLE = 1000,
    DRIVER_INVALID_ACS_REQUEST = 1001,
    DRIVER_ACS_HANDLE_REJECTION = 1002,
    DRIVER_INVALID_CLASS_REJECTION = 1003,
    :
}
```
DRIVER_GENERIC_REJECTION = 1004,
DRIVER_RESOURCELIMITATION = 1005,
DRIVER_ACSHANDLE_TERMINATION = 1006,
DRIVER_LINK_UNAVAILABLE = 1007
} ACSUniversalFailure_t;

- Driver errors

  Error values in this category indicate that the driver detected an error. These errors include one of the following specific error values:

- Driver Duplicate ACSHandle

  TheACSHandle given for an ACSOpenStream request is already in use for a session. The already open session with the ACSHandle remains open.

- DriverInvalid ACS Request

  The ACS message contains an invalid or unknown request. The request is rejected.

- Driver ACS Handle Rejection

  A CSTA request was issued with no prior ACSOpenStream request. The request is rejected.

- Driver Invalid Class Rejection

  The driver received a message containing an invalid or unknown message class. The request is rejected.

- Driver Generic Rejection

  The driver detected an invalid message for something other than message type or message class. This is an internal error and should be reported.

- Driver Resource Limitation

  The driver did not have adequate resources (that is, memory, etc.) to complete the requested operation. This is an internal error and should be reported.

- Driver ACSHandle Termination

  Due to problems with the link to the switch, the driver has found it necessary to terminate the session with the given ACSHandle. The session will be closed, and all outstanding requests will terminate.

- Driver Link Unavailable

  The driver was unable to open the new session because no link was available to the PBX. The link may have been placed in the BLOCKED state, or it may have been taken off line.
CSTA Messaging Interface

CSTA request, response and event messages transported across the TSDI are presented in a C structure format. Each API call and Event that needs to be passed across the TSDI maps directly to a C structure defined in cstadefs.h. This C structure will be passed in the message block portion of the buffer. A driver writer needs only to map the correct C structure, based on the messageClass and messageType, onto the message block (whose location is indicated by the messageOffset field) to access the information contained in the CSTA message.

There are five categories of CSTA API calls and Events:

- Switching Function Services
- Status Reporting Services
- CSTA Snapshot Services
- CSTA Computing Function Services
- CSTA Escape/Maintenance Services

Each of these categories has associated API calls and events that are illustrated below. The tables indicate the function name of the CSTA API call or the CSTA Event name, its associated C structure name (a full C structure definition is included in cstadefs.h), the message class and type of the C structure, and the associated C structure name of the confirmation event.

The naming conventions for the items in the tables are as follows:

<table>
<thead>
<tr>
<th>csta&lt;function name&gt;</th>
<th>csta API function call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTA&lt;function name&gt;Event</td>
<td>CSTA Event (request, confirmation or unsolicited)</td>
</tr>
<tr>
<td>&lt;function name&gt;_t</td>
<td>C structure name corresponding to a CSTA API call, confirmation event or unsolicited event</td>
</tr>
<tr>
<td>&lt;define name&gt;</td>
<td>message type name</td>
</tr>
</tbody>
</table>

Request-Response Protocol

An open ACS stream for CSTA or OA&M is used to exchange messages that represent a request-response protocol between the PBX Driver and the client application. The API function calls (CSTA or OA&M) map into requests sent in message format on the stream. Each message includes the same DC block as
the "open request" message contained: messageOffset, messageLength, messageClass, messageType, privateOffset, privateLength, invokeID, monitorCrossRefID, and a class_of_service.

The combination of messageType and messageClass map to a specific API function call, (CSTA, ACS or OA&M) and the invokeID identifies a specific API function call invocation. The PBX Driver must include the sessionID and the invokeID from the request message in the response sent back to the client application. The messageClass and messageType sent back in the response message to the client request must correspond to an appropriate response message type (see the “ACS Messaging Interface” on page 3-22, “CSTA Messaging Interface” on page 3-28, and “OA&M Interface” on page 3-41). This response message is mapped to a confirmation (or failure) event for the client application at the API.

Unsolicited events may also be generated by the PBX Driver for the client (CSTA or OA&M). The PBX Driver must include the sessionID, the messageClass, and the messageType in the event message sent to the client application. An invokeID is not used for events that are originated by the PBX Driver.

Requests may also be generated by the PBX Driver for the client (CSTA or OA&M). The PBX Driver must include the sessionID, the messageClass, and the messageType in the request sent to the client application. An invokeID is used for requests (except for Routing Requests see “CSTA Computing Function Services” on page 3-34) that are originated by the PBX Driver. These request messages are mapped into events for the Client at the API (CSTA or OA&M). The client must call an API function (CSTA or OA&M) to generate a response back to the PBX Driver. The response message always contains a sessionID, and an appropriate messageClass and messageType and possibly an invokeID.

Processing CSTA Messages

A Driver must respond in one of three ways for each CSTAREQUEST it receives:

Send back the appropriate CSTACONFIRMATION message indicating the requested operation has been initiated. Refer to [ECMA-179] for a description/definition of what each confirmation event means.

Send back a CSTAUniversalFailureConfEvent message indicating a CSTA type of failure and that the requested operation has been failed. Refer to [ECMA-179] for a description/definition of what each confirmation event means.

Send back an ACSFailureConfEvent messages indicating an ACS type of failure (problem maintaining the ACS Stream) and that the request operation has been failed. Refer to [TSAPI - Chapter 4] for a list of failure codes for an ACSFailureConfEvent that a Driver is allowed to use.
CSTA Control Services Functions

CSTA Control Service Functions allow an application to determine which set of CSTA functionality a Driver supports.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstaGetAPICaps()</td>
<td>CSTAGetAPICaps_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_GETAPI_CAPS</td>
<td>CSTAGetAPICapsConfEvent_t</td>
</tr>
<tr>
<td>CSTAGetAPICapsConfEvent</td>
<td>CSTAGetAPICapsConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_GETAPI_CAPS_CONF</td>
<td>NA</td>
</tr>
</tbody>
</table>

CSTA Security Services Functions

CSTA Security Services functions allow an application to determine which set of devices can be controlled as defined by the Telephony Server.

**NOTE:**
These messages are always handled by the Telephony Server and are never sent to a Driver and are listed here only for completeness; that is, every API call in [TSAPI] is listed in this document to clarify which of these messages are for the Driver and which are for the Tserver.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstaGetDeviceList()</td>
<td>CSTAGetDeviceList_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_GETDEVICE_LIST</td>
<td>CSTAGetDeviceListConfEvent_t</td>
</tr>
<tr>
<td>CSTATGetDeviceListConfEvent</td>
<td>CSTATGetDeviceListConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_GETDEVICE_LIST_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaQueryCallMonitor()</td>
<td>CSTATQueryCallMonitor_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_QUERY_CALL_MONITOR</td>
<td>CSTATQueryCallMonitorConfEvent_t</td>
</tr>
<tr>
<td>CSTATQueryCallMonitorConfEvent</td>
<td>CSTATQueryCallMonitorConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_QUERY_CALL_MONITOR_CONF</td>
<td>NA</td>
</tr>
</tbody>
</table>

NA - Not available

Switching Function Services

This section illustrates the functions and events of CSTA switching function services and their associated C structures that are presented across the TSDI. Switching function services are Telephony Services that operate on calls and activate switch-related features that are associated with the user desktop telephone or any other device defined by the switching domain. See [ECMA-179] for more details on the switching function services; see TSAPI for more information on the associated CSTA API C Language function calls and events.

Issue 1 — December 2002
This section defines Telephony Services that deal with basic call control for the desktop or call center environments. These functions provide services that allow client applications to:

- establish, control, and "tear down" calls at a device or within the switch,
- answer incoming calls into a device, and
- activate/deactivate features and capabilities supported by the switch or the server.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTAUniversalFailureConfEvent</td>
<td>CSTAUniversalFailureConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_UNIVERSAL_FAILURE_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaAlternateCall()</td>
<td>CSTAAlternateCall_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_ALTERNATE_CALL</td>
<td>NA</td>
</tr>
<tr>
<td>CSTAAlternateCallConfEvent</td>
<td>CSTAAlternateCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_ALTERNATE_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaAnswerCall()</td>
<td>CSTAAnswerCall_t</td>
<td>CSTAANSWER_CALL</td>
<td>CSTA_ANSWER_CALL</td>
<td>CSTAAnswerCallConfEvent t</td>
</tr>
<tr>
<td>CSTAAnswerCallConfEvent</td>
<td>CSTAAnswerCallConfEvent_t</td>
<td>CSTAANSWER_CALL_CONF</td>
<td>CSTA_ANSWER_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaCallCompletion()</td>
<td>CSTACallCompletion_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CALL_COMPLETION</td>
<td>CSTACallCompletionConfEvent t</td>
</tr>
<tr>
<td>CSTACallCompletionConfEvent</td>
<td>CSTACallCompletionConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CALL_COMPLETION_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaClearCall()</td>
<td>CSTAClearCall_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CLEAR_CALL</td>
<td>CSTAClearCallConfEvent t</td>
</tr>
<tr>
<td>CSTAClearCallConfEvent</td>
<td>CSTAClearCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CLEAR_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaClearConnection()</td>
<td>CSTAClearConnection_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CLEAR_CONNECTION</td>
<td>CSTAClearConnectionConfEvent t</td>
</tr>
<tr>
<td>CSTAClearConnectionConfEvent</td>
<td>CSTAClearConnectionConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CLEAR_CONNECTION_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaConferenceCall()</td>
<td>CSTAConferenceCall_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CONFERENCE_CALL</td>
<td>CSTAConferenceCallConfEvent t</td>
</tr>
<tr>
<td>CSTAConferenceCallConfEvent</td>
<td>CSTAConferenceCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CONFERENCE_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaConsultationCall()</td>
<td>CSTAConsultationCall_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CONSULTATION_CALL</td>
<td>CSTAConsultationCallConfEvent t</td>
</tr>
<tr>
<td>CSTAConsultationCallConfEvent</td>
<td>CSTAConsultationCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CONSULTATION_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaDeflectCall()</td>
<td>CSTADeflectCall_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_DEFLECT_CALL</td>
<td>CSTADeflectCallConfEvent t</td>
</tr>
<tr>
<td>CSTADeflectCallConfEvent</td>
<td>CSTADeflectCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_DEFLECT_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>Function</td>
<td>Request Name</td>
<td>Request Type</td>
<td>Event Type</td>
<td>Confirmation Type</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>--------------</td>
<td>------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>cstaGroupPickupCall()</td>
<td>CSTAGroupPickupCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_GROUP_PICKUP_CALL</td>
<td>CSTAGroupPickupCallConfEvent_t</td>
</tr>
<tr>
<td>CSTAGroupPickupCallConfEvent</td>
<td>CSTAGroupPickupCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_GROUP_PICKUP_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaHoldCall()</td>
<td>CSTAHoldCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_HOLD_CALL</td>
<td>CSTAHoldCallConfEvent_t</td>
</tr>
<tr>
<td>CSTAHoldCallConfEvent</td>
<td>CSTAHoldCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_HOLD_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaMakeCall()</td>
<td>CSTAMakeCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_MAKE_CALL</td>
<td>CSTAMakeCallConfEvent_t</td>
</tr>
<tr>
<td>CSTAMakeCallConfEvent</td>
<td>CSTAMakeCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_MAKE_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaMakePredictiveCall()</td>
<td>CSTAMakePredictiveCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_MAKE_PREDICTIVE_CALL</td>
<td>CSTAMakePredictiveCallConfEvent_t</td>
</tr>
<tr>
<td>CSTAMakePredictiveCallConfEvent</td>
<td>CSTAMakePredictiveCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_MAKE_PREDICTIVE_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaPickupCall()</td>
<td>CSTAPickupCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_PICKUP_CALL</td>
<td>CSTAPickupCallConfEvent_t</td>
</tr>
<tr>
<td>CSTAPickupCallConfEvent</td>
<td>CSTAPickupCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_PICKUP_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaReconnectCall()</td>
<td>CSTAReconnectCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_RECONNECT_CALL</td>
<td>CSTAReconnectCallConfEvent_t</td>
</tr>
<tr>
<td>CSTAReconnectCallConfEvent</td>
<td>CSTAReconnectCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_RECONNECT_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaRetrieveCall()</td>
<td>CSTARetrieveCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_RETRIEVE_CALL</td>
<td>CSTARetrieveCallConfEvent_t</td>
</tr>
<tr>
<td>CSTARetrieveCallConfEvent</td>
<td>CSTARetrieveCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_RETRIEVE_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaTransferCall()</td>
<td>CSTATransferCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_TRANSFER_CALL</td>
<td>CSTATransferCallConfEvent_t</td>
</tr>
<tr>
<td>CSTATransferCallConfEvent</td>
<td>CSTATransferCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_TRANSFER_CALL_CONF</td>
<td>NA</td>
</tr>
</tbody>
</table>
Status Reporting Services

This section illustrates the functions and events of the CSTA Status Reporting Services and their associated C structures that are presented across the TSDI. Status Reporting Services encompass the function calls and events that have to do with unsolicited event messages coming from the Telephony Server. Unsolicited event messages can be generated as a result of external telephony activity on the switch/device or activity generated by the users at the physical telephone instrument. The status reporting request function allows the applications to turn on or turn off status event reporting for an associated CSTA device (for example, a desktop telephone). See [ECMA-179] for more details on the status reporting services and see TSAPI for more information on the associated CSTA API C Language function calls and events.

CSTA Snapshot Services

This section describes the CSTA Snapshot Services available to query the current state of a call or a device within the switching domain accessible by the application using this API. These services provide the application with specific information about a call or a device object by requesting that the switch query the object to determine its state. The information provided by this service is a snapshot in time of the state of a call or device object. Due to the dynamic nature

### API Call or ACS Event Name

<table>
<thead>
<tr>
<th>C Language Structure Name</th>
<th>C Language Message Class</th>
<th>Message Type</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstaMonitorDevice( )</td>
<td>CSTAMonitorDevice_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_MONITOR_DEVICE</td>
</tr>
<tr>
<td>cstaMonitorCall( )</td>
<td>CSTAMonitorCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_MONITOR_CALL</td>
</tr>
<tr>
<td>cstaMonitorCallsViaDevice( )</td>
<td>CSTAMonitorCallsViaDevice_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_MONITOR_CALLS_VIA_DEVICE</td>
</tr>
<tr>
<td>CSTAMonitorConfEvent</td>
<td>CSTAMonitorConfEvent_t</td>
<td>CSTA_CONFIRMATION</td>
<td>CSTA_MONITOR_CONF</td>
</tr>
<tr>
<td>cstaChangeMonitorFilter()</td>
<td>CSTAChangeMonitorFilter_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_CHANGE_MONITOR_FILTER</td>
</tr>
<tr>
<td>CSTAChangeMonitorFilterConfEvent</td>
<td>CSTAChangeMonitorFilterConf_t</td>
<td>CSTA_CONFIRMATION</td>
<td>CSTA_CHANGE_MONITOR_FILTER_CONF</td>
</tr>
<tr>
<td>cstaMonitorStop()</td>
<td>CSTAMonitorStopStop_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_MONITOR_STOP</td>
</tr>
<tr>
<td>cstaChangeMonitorFilter()</td>
<td>CSTAChangeMonitorFilter_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_CHANGE_MONITOR_FILTER</td>
</tr>
<tr>
<td>CSTAChangeMonitorFilterConfEvent</td>
<td>CSTAChangeMonitorFilterConf_t</td>
<td>CSTA_CONFIRMATION</td>
<td>CSTA_CHANGE_MONITOR_FILTER_CONF</td>
</tr>
<tr>
<td>CSTAMonitorEnded</td>
<td>CSTAMonitorEnded_t</td>
<td>CSTA_UNSOLICITED</td>
<td>CSTA_MONITOR_ENDED</td>
</tr>
</tbody>
</table>

NA - Not available
of calls and Connection States at devices, any snapshot information provided to the application may become outdated as time elapses. This can occur because of additional changes in the state of calls within the switching domain after the switch has completed the call or device query.

### CSTA Computing Function Services

This section illustrates the functions and events of the CSTA Computing Function Services and their associated C structures that are presented across the TSDI. Computing Services allow the client/server role between the application and the switch to be reversed where the application becomes the server for call routing requests being originated by the switch. Call routing allows the switch to pass any available call-related information to the application and request routing information for the call from the application. See [ECMA-179] for more details on the computing function services. Also refer to the Telephony Services API (TSAPI) for more information on the associated CSTA API C Language function calls and events.

### Routing Registration Functions and Events

This section describes the service requests and events used by an application to register with the Telephony Server as a call routing server for a specific routing device.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstaSnapShotCallReq( )</td>
<td>CSTASnapshotCall_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_SNAPSHOT_CALL</td>
<td>CSTASnapshotCallConfEvent_t</td>
</tr>
<tr>
<td>CSTASnapshotCallConfEvent</td>
<td>CSTASnapshotCallConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_SNAPSHOT_CALL_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaSnapShotDeviceReq( )</td>
<td>CSTASnapshotDevice_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_SNAPSHOT_DEVICE</td>
<td>CSTASnapshotDeviceConfEvent_t</td>
</tr>
<tr>
<td>CSTASnapshotDeviceConfEvent</td>
<td>CSTASnapshotDeviceConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_SNAPSHOT_DEVICE_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaRouteRegisterReq( )</td>
<td>CSTARouteRegisterReq_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_ROUTE_REGISTER_REQ</td>
<td>CSTARouteRegisterReqConfEvent_t</td>
</tr>
<tr>
<td>CSTARouteRegisterReqConfEvent</td>
<td>CSTARouteRegisterReqConfEvent_t_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_ROUTE_REGISTER_REQ_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaRouteRegisterCancel( )</td>
<td>CSTARouteRegisterCancel_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_ROUTE_REGISTER_CANCEL</td>
<td>CSTARouteRegisterCancelConfEvent_t</td>
</tr>
</tbody>
</table>

**Issue 1 — December 2002**
Routing Functions and Events

This section defines the CSTA call routing services that can be utilized for application-based call routing within the switching domain. Calls that are routed using these services are queued at the routing device until the application provides a destination for the call or a timeout condition occurs at the call routing queue within the switching domain. Application-based call routing is handled using a routing dialogue between a routing client (the driver/switch) and the routing server (the application). This dialogue is accomplished using the functions and events defined in this section.

These functions and events can be used once the application has requested and has been granted call routing capabilities for a specific device or Telephony Server (see “Routing Registration Functions and Events” on page 3-34 for more details on registering as a routing server). A CSTARouteRequestExtEvent (or CSTARouteRequestEvent) will be sent to the application for each call that requires a routing destination from the routing server, that is, the application. The route request response is issued by the application using the cstaRouteSelectInv() (or cstaRouteSelect()) function that provides the switch with the appropriate destination for the call (for example, a destination address - device ID/telephone number). Once the routing information reaches the switch, it will attempt to route the call to the destination provided by the application in the cstaRouteSelectInv() function. The application should check the CSTARouteEndEvent and/or the CSTARouteUsedExtEvent (or CSTARouteUsedEvent) to ensure that the route request has been completed by the switch. If a routing destination is invalid within the switching domain the driver/switch may request additional routing information (a different destination than the one provided previously) using the CSTARouteEvent.

Register Request ID vs. Routing Cross-Reference ID

The routing services described in this document use two new handles (identifiers) to refer to different software objects within the Telephony Server. The register request identifier (routeRegisterReqID) is used to identify the specific routing session over which routing requests will be generated. This handle is specific to a routing device within the switch or to a specific ACS Stream in the case of the default routing server. The routeRegisterReqID will exist after the application successfully registers for routing services (cstaRouteRegisterReq( )) and until the registration is canceled (cstaRouteRegisterCancel( )).
Within a specific routing session (routeRegisterReqID) there may be many routing dialogs created by the driver/switch to identify the routing instance of a particular call. This routing dialog is established for the duration of the call routing dialog between the driver/switch and the routing server. The handle to this routing dialog is known as the routing cross-reference identifier (routingCrossRefID). This handle is valid after a new call arrives at the routing device and the driver/switch sends a CSTARouteRequestEvent. The routingCrossRefID specified in the route request event will be valid for the duration of the call routing dialog or until a route end event is sent by either the driver/switch or the application.

The routing cross-reference identifier (routingCrossRefID) will be unique within the same routing session (routeRegisterReqID). Some driver/switch implementations may provide the additional benefit of a unique routing cross-reference identifier across the entire switching domain regardless of the specific routing session. Routing session identifiers (routeRegisterReqID) will be unique within the same ACS Stream (sessionID).

Both the routeRegisterReqID and routingCrossRefID are generated by the Driver.

NOTE:
If a call is not successfully routed by the routing server, this does not necessarily mean that the call is cleared or not answered. Most switch implementations will have a default mechanism for handling a call at a routing device when the routing server has failed to provide a valid destination for the call.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTARouteRequestExtEvent†</td>
<td>CSTARouteRequestExtEvent_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_ROUTE_REQUEST_EXT</td>
<td>cstaRouteSelectInv( )</td>
</tr>
<tr>
<td>CSTARouteRequestEvent</td>
<td>CSTARouteRequestEvent_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_ROUTE_REQUEST</td>
<td>cstaRouteSelect( )</td>
</tr>
<tr>
<td>CSTAReRouteRequestEvent</td>
<td>CSTAReRouteRequestEvent_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_RE_ROUTE_REQUEST</td>
<td>cstaRouteSelect( ) or cstaRouteSelectInv()</td>
</tr>
<tr>
<td>cstaRouteSelectInv( )†</td>
<td>CSTARouteSelectInvRequest_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_ROUTE_SELECT_INV_REQUEST</td>
<td>NA</td>
</tr>
<tr>
<td>cstaRouteSelect( )</td>
<td>CSTARouteSelectRequest_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_ROUTE_SELECT_REQUEST</td>
<td>NA</td>
</tr>
<tr>
<td>CSTARouteUsedExtEvent†</td>
<td>CSTARouteUsedExtEvent_t</td>
<td>CSTAEVENTREPORT</td>
<td>CSTA_ROUTE_USED_EXT</td>
<td>NA</td>
</tr>
<tr>
<td>CSTARouteUsedEvent</td>
<td>CSTARouteUsedEvent_t</td>
<td>CSTAEVENTREPORT</td>
<td>CSTA_ROUTE_USED</td>
<td>NA</td>
</tr>
</tbody>
</table>
CSTA Escape/Maintenance Services

Two types of maintenance services are defined within the CSTA standard:

- Device status maintenance events that provide status information for device objects
- Bi-directional system status maintenance services that provide information on the overall status of the system

The device status events inform the application when a monitored device is placed in or out of service. When a device object is placed out of service, the application will be limited to monitoring the device (for example, cstaMonitorDevice() or cstaSnapshotDeviceReq()) and no active services are allowed. For example, a cstaMakeCall() service request is not allowed when the device is out of service). The device status events will include the CSTA association being used to monitor the device; that is, the monitorCrossRefID. The driver must enforce this limitation.

Escape Services: Application as Client

This section defines escape services for cases where the application is the service requester in the client/server relationship. The services include an escape service request, a confirmation event to the request, and an unsolicited escape service event that originates at the driver or switching domain.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstaEscapeService()</td>
<td>CSTAEscapeSvc_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_ESCAPE_SVC</td>
<td>CSTAEscapeSvcConfEvent_t</td>
</tr>
<tr>
<td>CSTAEscapeSvcConfEvent</td>
<td>CSTAEscapeSvcConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_ESCAPE_SVC_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>CSTAPrivateEvent</td>
<td>CSTAPrivateEvent_t</td>
<td>CSTAEVENTREPORT</td>
<td>CSTA_PRIVATE</td>
<td>NA</td>
</tr>
<tr>
<td>CSTAPrivateStatusEvent</td>
<td>CSTAPrivateStatusEvent_t</td>
<td>CSTAUNSOLICITED</td>
<td>CSTA_PRIVATE_STATUS</td>
<td>NA</td>
</tr>
</tbody>
</table>
Escape Service: Driver/Switch as the Client

This section defines escape services for cases where the driver/switch is the service requester in the client/server relationship. The services include an escape service request event, a confirmation function for the request, and an unsolicited escape service event that originates at the application domain.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTAEscapeSvcReqEvent</td>
<td>CSTAEscapeSvcReqEvent_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_ESCAPE_SVC_REQ</td>
<td>CSTAEscapeSvcConfEvent_t</td>
</tr>
<tr>
<td>cstaEscapeServiceConf()</td>
<td>CSTAEscapeSvcReqConf_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_ESCAPE_SVC_REQ_CONF</td>
<td>NA (No confirmation event required for this request message)</td>
</tr>
<tr>
<td>cstaSendPrivateEvent()</td>
<td>CSTASendPrivateEvent_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_SEND_PRIVATE</td>
<td>NA (No confirmation event required for this request message)</td>
</tr>
</tbody>
</table>

NA - Not available

Maintenance Services

This section identifies those events associated with the CSTA maintenance capabilities and the private event used as an escape mechanism to send implementation-specific unsolicited events.

Device Status

This section describes the CSTA Maintenance Services, which provide device status information. The device must be monitored by the application, with an active CSTA monitor association (for example, an active `monitorCrossRefId`), to receive this event. These events are unidirectional and always originate in the driver/switch domain.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTABackInServiceEvent</td>
<td>CSTABackInServiceEvent</td>
<td>CSTAUNSOLICITED</td>
<td>CSTA_BACK_IN_SERVICE</td>
<td>NA</td>
</tr>
<tr>
<td>CSTAOutOfServiceEvent</td>
<td>CSTAOutOfServiceEvent_t</td>
<td>CSTAUNSOLICITED</td>
<td>CSTA_OUT_OF_SERVICE</td>
<td>NA</td>
</tr>
</tbody>
</table>

NA - Not available
System Status: Application as the Client

This section defines the services that provide system-level status information to the application or the driver/switch. The System Status service is bi-directional and, thus, the client/server relationship can be reversed.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstaSysStatReq()</td>
<td>CSTAReqSysStat_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_REQ_SYS_STAT</td>
<td>CSTASysStatReqConfEvent_t</td>
</tr>
<tr>
<td>CSTASysStatReqConfEvent_t</td>
<td>CSTASysStatReqConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_SYS_STAT_REQ_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaSysStatStart()</td>
<td>CSTASysStatStart_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_SYS_STAT_START</td>
<td>CSTASysStatStartConfEvent_t</td>
</tr>
<tr>
<td>CSTASysStatStartConfEvent_t</td>
<td>CSTASysStatStartConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_SYS_STAT_START_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaSysStatStop()</td>
<td>CSTASysStatStop_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_SYS_STAT_STOP</td>
<td>CSTASysStatStopConfEvent_t</td>
</tr>
<tr>
<td>CSTASysStatStopConfEvent_t</td>
<td>CSTASysStatStopConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_SYS_STAT_STOP_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaChangeSysStatFilter()</td>
<td>CSTAChangeSysStatFilter_t</td>
<td>CSTAREQUEST</td>
<td>CSTA_CHANGE_SYS_STAT_FILTER</td>
<td>CSTAChangeSysStatFilterConfEvent_t</td>
</tr>
<tr>
<td>CSTAChangeSysStatFilterConfEvent_t</td>
<td>CSTAChangeSysStatFilterConfEvent_t</td>
<td>CSTACONFIRMATION</td>
<td>CSTA_CHANGE_SYS_STAT_FILTER_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>CSTASysStatEvent</td>
<td>CSTASysStatEvent_t</td>
<td>CSTAEVENTREPORT</td>
<td>CSTA_SYS_STAT</td>
<td>NA</td>
</tr>
<tr>
<td>CSTASysStatEndedEvent</td>
<td>CSTASysStatEndedEvent_t</td>
<td>CSTAEVENTREPORT</td>
<td>CSTA_SYS_STAT_ENDED</td>
<td>NA</td>
</tr>
</tbody>
</table>
## System Status: Driver/Switch as the Client

This section defines the services that provide system-level status information to the driver/switch from the application. The System Status service is bi-directional; thus, the client/server relationship can be reversed.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>C Language Confirmation Event Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSTASysStatReqEvent_ t</td>
<td>CSTASysStatReqEvent_ t</td>
<td>CSTAUNSOLICITED</td>
<td>CSTA_SYS_STAT_REQ</td>
<td>cstaSysStatReqConf( )</td>
</tr>
<tr>
<td>cstaSysStatReqConf()</td>
<td>CSTAReqSysStatConf_ t;</td>
<td>CSTAREQUEST</td>
<td>CSTA_REQ_SYS_STAT_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>cstaSysEventSend( )</td>
<td>CSTASysEventSend_ t</td>
<td>CSTAREQUEST</td>
<td>CSTA_SYS_STAT_EVENT_SEND</td>
<td>NA (No confirmation event required for this request message )</td>
</tr>
</tbody>
</table>
OA&M Interface

The Tserver will serve as a pass through for driver OA&M facilities so that driver vendors can implement a client-based user interface for driver OA&M, and use the Tserver as a transport mechanism to the driver. To use the Tserver to route driver OA&M messages a driver must register with the Tserver via the tdidriverRegister() routine for OA&M functionality. The Tserver will treat messages received from the client application as a block of data and pass the message directly to the driver that has registered for the OA&M services. The data contained within the message block is to be defined by the driver writer and is specific to each vendor's driver.

A client that wants to open an OA&M session with a driver must call acsOpenStream(). The authentication of the client to perform OA&M functions for the driver will be accomplished via a login ID and a password. The login ID provided must be administered on the Tserver to perform OA&M for the registered driver. Blocks of data containing driver defined OA&M requests can be sent to the driver via tsrvDriverRequest(). The corresponding confirmation event for this request is TSrvDriverOAMConfEvent. The driver may also send unsolicited OA&M events using TSrvDriverOAMEvent. An OA&M session must be terminated via the acsCloseStream() routine.

OA&M Interface Control Services

This section illustrates the functions and events of the OA&M Interface Control Services and their associated C structures that are presented across the TSDI. OA&M ICS functions deal with the characteristics of the API (for example, opening and closing the OA&M interface). They open, initialize, close, and manage a virtual communication channel (OA&M stream) with any advertised OA&M registered service. See Chapter 6 for a specification of the OA&M API C Language function calls and events.

<table>
<thead>
<tr>
<th>API Call or ACS Event Name</th>
<th>C Language Structure Name</th>
<th>Message Class</th>
<th>MessageType</th>
<th>Confirmation Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsrvDriverRequest()</td>
<td>NA</td>
<td>TDRVRREQUEST</td>
<td>TSRV_DRIVEROAM_REQ</td>
<td>TSrvDriverOA MConfEvent</td>
</tr>
<tr>
<td>TSRVDriverOAMConfEvent</td>
<td>TSRVDriverOAM ContEvent_t</td>
<td>TDRVRCONFIRMATION</td>
<td>TSRV_DRIVEROAM_CONF</td>
<td>NA</td>
</tr>
<tr>
<td>TSRVDriverOAMEvent</td>
<td>TSRVDriverOAM Event_t</td>
<td>TDRVRUNSOLICITED</td>
<td>TSRV_DRIVEROAM</td>
<td>NA</td>
</tr>
</tbody>
</table>
TSAPI Version Control

TSAPI Version Control is the mechanism by which an application can negotiate the version or versions of TSAPI that it desires to use. Version control ensures that applications written to earlier versions of TSAPI are compatible with newer versions of client libraries, Tservers, and PBX Drivers, and that applications written to later versions of TSAPI can operate with older versions of client libraries, Tservers, and PBX Drivers.

To implement version control in TSAPI, an application must supply a list of API versions it is willing to accept in the apiVer parameter of the acsOpenStream() function. This parameter is a string with a format that allows one or more versions to be specified. The string begins with the characters TS followed by a numeric ASCII encoding of one or more version numbers. The - (hyphen) character can be used to specify a range of versions and the : (colon) character to separate a list of versions. For example, the string:

“TS1-3:5” specifies that the application is willing to accept API versions 1, 2, 3, or 5.

As the open stream request is processed, each system component checks whether it can support the listed versions of the API (that is, TSAPI protocol versions). The Tserver performs the check on behalf of the PBX Driver, since the PBX Driver supplied the list of versions it supports during driver registration. If any component cannot support a given version, then that version is removed from the list before being passed to the next component. The highest API version remaining is returned to the application in the resulting ACSOpenStreamConfEvent and this API version will remain in effect for the life of the stream. If no compatible API version is found, then the open stream request will fail and the Tserver returns an ACSUniversalFailureConfEvent.

By the time the PBX Driver receives the ACSOpenStream message, the number of versions in the apiVer string will have been reduced to just one. The PBX Driver must either accept this version or reject it; it cannot change it. The PBX Driver should return the same value in the apiVer of the ACSOpenStreamConfEvent as it received in the acsOpenStream() message, since a Release 2 Tserver will overwrite the string appropriately before returning it to the application, but a Release 1 Tserver will return the string unmodified.

The selected API version is returned to the application in the apiVer field of the ACSOpenStreamConfEvent and has the following format. The string begins with the characters ST (the S and T are intentionally reversed to prevent false API version indications from Release 1 PBX Drivers) followed by a single API version number (for example, ST2).

An API version is defined by a collection of API function calls and events. New API functions will always be given new and unique names, and new event reports will be assigned new and unique event type values. The association between API functions, events, and API versions is documented in the TSAPI definition. API
versions are defined on a per-service type basis. That is, CSTA streams will use
a distinct version numbering scheme from OAM streams. For Release 2, only
CSTA streams require a new version number (for the new CSTA Computing
Function Services functions and events), since there are no changes to the OAM
messages. Release 2 of the T server supports CSTA stream versions TS1 and
TS2, and OAM stream version TS1. Note that TSAPI API protocol versions and
Telephony Services release numbers are distinct. In the future, CSTA API version
numbers and Telephony Services release numbers may not map one-to-one as
they do for Releases 1 and 2.

As mentioned previously, a new version of the driver registration function,
tdiDriverRegister(), has been defined in TSDI Version 2. A previously
unused parameter (channel_number), has been changed to the protocol_
descriptor parameter. The protocol_descriptor parameter informs the T server
which TSAPI versions the PBX Driver supports. This parameter is a type
TDIProtocol_t (unsigned long) where each bit corresponds to a specific TSAPI
version. Bit 0 (LSB) corresponds to TSAPI version 1, bit 1 corresponds to TSAPI
version 2, and so on. Multiple version support is indicated by OR-ing the
appropriate bits together. Constant definitions for each version are provided in the
header file tdi.h supplied in the PBX Driver SDK.

The version parameter of tdiDriverRegister() must indicate version
TSDL_VERSION_2 for the T server to interpret the protocol_descriptor
parameter correctly. If the PBX Driver registers with the version parameter set to
TSDL_VERSION, the T server will ignore the protocol_descriptor parameter and
assume that the driver only supports the version 1 TSAPI protocol.

Release 1 client libraries and Tservers do not check or alter the open stream
request apiVer field. As a result, a PBX Driver registered with a Release 1
Tserver should ignore the apiVer field. The apiVer field should also be ignored if
it does not adhere to the proper format (for example, does not begin with TS),
which can happen when a stream is being opened by an R1 application and R1
client library. In these cases, the PBX Driver should process the open stream
request normally but assume the version 1 TSAPI protocol.
Private Data Definition

Private data may be sent by the client application or Driver with (nearly) every message defined in the [TSAPI]. Private data must always be sent by the client application (via the API interface) and received by the Driver across the TSDI or sent by the Driver across the TSDI and received by the client application as a PrivateData_t as defined in acs.h. The PrivateData has the following structure:

typedef struct PrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[1]; //actual length determined
    //by application
} PrivateData_t;

The vendor field can be filled in any way the Driver and Application define. The length field must indicate the size of the character array that starts at the pointer data. The length field must be set, because this indicates to the transport layers how many bytes to transmit. The data field should be interpreted as an array of characters of size length that starts at the char pointer data. The format of the character array is defined entirely by the Driver and the application.

Private Data Version Negotiation

Beginning with Release 2, a standard mechanism has been defined for negotiating the private data version to be used by an application over a client session or stream. The standard mechanism allows applications to easily determine the (vendor) type and version(s) of private data a PBX Driver supports, and to reduce traffic on the LAN when private data is not used by the application. Although similar, private data version negotiation should be thought of as mostly independent of TSAPI Version Control.

The vendor string and its associated versions are selected by each PBX Driver vendor. Over time, there may exist several versions of a given vendor’s private data. An application opening a stream to a PBX Driver must be able to tell the driver which (vendor) types and versions of private data it can support. In turn, the PBX Driver must determine if it can support one or more of those (vendor) types and versions and notify the application as to which one it has selected. The result of the negotiation determines the private data (vendor) type and version for the life of the stream.

The TSAPI open stream request is used to negotiate private data version. If an application wishes to use private data, it must supply a list of private data vendors and associated versions it can support in the open stream request. This list is itself contained in the private data passed with the request.

The format of the open stream private data is as follows: The vendor field of the PrivateData_t structure should be set to the string VERSION. This identifies the private data as a version negotiation message. The data portion of the structure
should contain a 1 byte encoding discriminator that should be set to the manifest constant `PRIVATE_DATA_ENCODING` (0) followed by a NULL-terminated ASCII string that contains a list of private data (vendor) types and versions. The string contains a list of (vendor) type and version pairs separated by the # character. Each (vendor) type and version is also separated by a #. The version is encoded as ASCII numeric digits, where the - character can be used to specify a range of versions and the : to delimit a list of versions.

The following is an example of a version negotiation string:

"VENDOR1#1-3#VENDOR2#1:3-5"

The string indicates that the application can support VENDOR1 private data versions 1, 2, and 3, and VENDOR2 private data versions 1, 3, 4, and 5. The `data` member of `PrivateData_t` would appear as pictured below for this private data negotiation:

```
0 'V' 'E' 'N' 'D' 'O' 'R' '1' '#' '1' '-' '3' '#' 'V' 'E' 'N' 'D' 'O' 'R' '2' '#' '1' ':' '3' '-' '5' 0
```

Once the PBX Driver has selected a common private data (vendor) type and version, it responds with an `ACSOpenStreamConfEvent` that identifies (again within the private data) the selected vendor and version. The format for the private data sent with the `ACSOpenStreamConfEvent` is as follows: The `vendor` field of the `PrivateData_t` structure is set to the selected vendor string. The `data` portion of the structure contains a 1 byte encoding discriminator that should be set to `PRIVATE_DATA_ENCODING` followed by an ASCII string that identifies the single private data version selected. The PBX Driver should select the highest common version of private data that both it and the application can support.

An application that does not use private data should not add private data to the open stream request. Absence of the private data indicates to the PBX Driver that the application is not using private data. The PBX Driver should not send private data to the application in this case, except as noted below.

The PBX Driver must maintain support for Release 1 applications, which do not support private data version negotiation. An application requesting a version 1 API stream (that is, TSAPI protocol version 1) without private data in the open stream request should still receive version 1 private data. However, applications requesting and obtaining a version 2 or higher API stream without private data in the open stream should not be sent private data by the PBX Driver. This eliminates waste of LAN resources when the application does not use the private data.

A PBX Driver that does not support private data need not examine the `ACSOpenStream` message for a private data negotiation message nor generate a reply.
Error Log Interface

The T server exports a standard function call interface to the PBX Driver so that
the Driver can log errors to the Telephony Server error log file. When a PBX Driver
uses the tdiLogError() function, the errors reported by the Driver and the
T server will have a uniform appearance in the error log.

The error log interface provides the following six severity levels through the level
parameter:

- **TRACE** This level is used for logging a trace message (for
debugging transient problems).
- **CAUTION** This level is used to log a nonservice-affecting
software condition that is not fatal.
- **AUDIT_TRAIL** This level is used to log important (normal) events:
driver loaded, link reset, etc.
- **WARNING** This level is used to indicate a problem that in of
itself is not service-affecting, but indicates a
condition that may become a problem (for example,
low resources).
- **ERROR** This level is used to log a service-affecting
condition that is not fatal.
- **FATAL** This level is used to log a fatal problem with the
logging module.

There are destinations other than the error log that can be enabled for each
severity level. These destinations can be set through the Tserver's OA&M
application. Refer to the manual page for tdiLogError() for a description of
these settings.

The error message itself is specified via a printf-like format string in the format
parameter and a variable number of parameters.

See “tdiLogError()” on page 5-19 for the manual page for this interface.
TSDI Coding

Initializing the Driver with the Tserver

This section contains coding examples for registering a driver with the Tserver, sending the Tserver a sanity message every minute, uses of the tdiLogError() function, and, finally, unregistering the driver during an unload.

main()

This is the main routine of the driver. It performs all initialization including registering with the Tserver and the Traffic Measurements Utility. After this function executes the driver should be ready to handle client requests.

```c
main()
{
    int rc;

    // Perform initialization here
    :
    :
    :

    /*
    * Register the driver for CSTA Services
    */
    CstaTDIHandle = tdiDriverRegister(SERVICE_NAME,
        DRIVER_NAME,
        TDI_ST_CSTA,
        TDI_CSTA_VERSION_1|TDI_CSTA_VERSION_2,
        VENDOR_NAME,
        TSDI_VERSION_2,
        TDI_CSTA_SECURITY,
        NULL);

    if (CstaTDIHandle < 0)
    {
        tdiLogError(DRIVER_NAME,
            FATAL,
            LOC_CODE,
            ERR_NO,
```
"Call to tdiDriverRegister failed:%d",
CstaTDIHandle);

return;

/*
 * Register the CSTA driver with the
 * Traffic Measurements Utility
 */
registerTraffic(CstaTDIHandle);

// Spin off a thread that will send the sanity message
// to the Tserver every minute.

unloadDriver()

This function unregisters with the Traffic Measurements Utility and then the TSDI. After unregistering it cleans up all of its resources and threads.

unloadDriver()
{
    int   rc;

    /*
     * Unregister the CSTA driver with the
     * Traffic Measurements Utility
     */
    unregisterTraffic(CstaTDIHandle);

    /*
     * Unregister the CSTA Driver
     */
    if ((rc = tdiDriverUnregister(CstaTDIHandle)) != TDI_SUCCESS)
    {
        tdiLogError(
            DRIVER_NAME,
            FATAL,
            LOC_CODE,
            ERR_NO,
            "tdiDriverUnRegister failed : %d",
            rc);
    }

    //Clean up any other remaining resources
}

Issue 1 — December 2002
sanity()

This is the sanity thread that sends the I’m Here message to the T server once every minute.

```c
void sanity(TDIHandle_t *tdiHandle)
{
    while(TRUE)
    {
        tdiDriverSanity(tdiHandle);
        delay(60000);
    }
}
```

Traffic Measurement Functions

This section contains coding examples for functions that register the driver with the Traffic Measurements Utility, report traffic measurements, support the callback function, and manage the traffic state for the driver. The previous section shows the main thread calling the registerTraffic() function and the unload routine calling the unregisterTraffic() function. The calls to reportTraffic() should be placed throughout the code where the driver receives messages from the PBX and where the driver sends messages to the PBX. Remember to report one message at a time.

Traffic Registration

This function registers the driver with the Traffic Measurements Utility.

```c
int registerTraffic ( TDIHandle_t driverID )
{
    int cc;
    int rc;
    int trafficState;

    trafficState = getTrafficState (driverID);
    if ( trafficState != NOT_REGISTERED_WITH_TDI )
    {
        /*
        * only attempt to register an
        * unregistered driver
        */
        return 0;
    }

    tdiLogError ( DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
        "Registering for traffic measurements ");

    cc = tdiTrafficRegister(driverID, TDI_PBX_DRIVER, trafficCB);

    switch ( cc )
    {
        case TDI_SUCCESS:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Traffic registration successful ");
            break;
        }
        case TDI_INVALID_DRIVER:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Invalid driver ID ");
            break;
        }
        case TDI_ALREADY_REGISTERED:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Driver already registered ");
            break;
        }
        case TDI_UNSUPPORTED_DRIVER:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Driver not supported ");
            break;
        }
        case TDI_INVALID_TYPE:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Invalid type ");
            break;
        }
        case TDI_INVALID_USAGE:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Invalid usage ");
            break;
        }
        case TDI_INVALID_ARGS:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Invalid arguments ");
            break;
        }
        case TDI_INVALID_POINTER:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Invalid pointer ");
            break;
        }
        case TDI_INVALID_STATE:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Invalid state ");
            break;
        }
        case TDI_INVALID_OPERATION:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Invalid operation ");
            break;
        }
        case TDI_INVALID_HANDLE:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Invalid handle ");
            break;
        }
        case TDI_UNKNOWN_ERROR:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Unknown error ");
            break;
        }
        default:
        {
            tdiLogError(DRIVER_NAME, AUDIT_TRAIL, LOC_CODE, ERR_NO,
                "Unknown error ");
            break;
        }
    }
    return cc;
}
```
AUDIT_TRAIL,
LOC_CODE,
"Successfully registered for traffic measurements -- measurements are ON"
);

setTrafficState ( TDI_TRAFFIC_ENABLED, driverID );
rc = 0;
break;
}

} case TDI_ERR_TRAFFIC_OFF:
{

tdiLogError(
DRIVER_NAME,
AUDIT_TRAIL,
LOC_CODE,
"Successfully registered for traffic measurements -- measurements are OFF"
);

setTrafficState ( TDI_TRAFFIC_DISABLED, driverID );
rc = 0;
break;
}

} default:
{

tdiLogError(DRIVER_NAME,
" Error %ld occurred while registering for traffic measurements - measurements are UNREGISTERED", 
cc);

/*
 * no need to set traffic state since we wouldn't have gotten here without already
 * being NOT_REGISTERED_WITH_TDI
 */

rc = -1;
break;
}
}

return rc;
}

Traffic Unregistration

This function unregisters the driver with the Traffic Measurements Utility.

int unregisterTraffic ( TDIHandle_tdriverID )
{

int cc, rc;

if ( getTrafficState (driverID) == NOT_REGISTERED_WITH_TDI )
{
/*
 * only attempt to unregister an ENABLED or DISABLED registered driver
 */

return 0;

}
Maintaining Traffic State

These functions set and return the current Traffic Measurement State for this driverID.

```c
int getTrafficState ( TDIHandle_t driverID )
{
    return TrafficState[driverID];
}

void setTrafficState ( int trafficState, TDIHandle_t driverID )
{
    switch ( trafficState )
    {
        case TDI_TRAFFIC_ENABLED:
        {
            if ( trafficState != TrafficState[driverID] )
            {
                tdiLogError(DRIVER_NAME, TRACE, LOC_CODE, ERR_NO, "Traffic measurements are ENABLED");
                TrafficState[driverID] = TDI_TRAFFIC_ENABLED;
            }
        }
    }
    return rc;
}
```
break;

}  

}  

}  

break;

}

}  

//case TDI_TRAFFIC_DISABLED:
{
if ( trafficState != TrafficState[driverID] )
{
 tdiLogError(DRIVER_NAME, 
 TRACE, 
 LOC_CODE, 
 ERR_NO, 
 "Traffic measurements are 
 DISABLED" );

 TrafficState[driverID] = 
 TDI_TRAFFIC_DISABLED;
}
break;
}

case NOT_REGISTERED_WITH_TDI:
{
if ( trafficState != TrafficState[driverID] )
{
 tdiLogError(DRIVER_NAME, 
 TRACE, 
 LOC_CODE, 
 ERR_NO, 
 "Traffic measurements are 
 UNREGISTERED" );

 TrafficState[driverID] = 
 NOT_REGISTERED_WITH_TDI;
}
break;
}

default:
{
 tdiLogError(DRIVER_NAME, 
 ERROR, 
 LOC_CODE, 
 ERR_NO, 
 "Invalid traffic measurement state%ld", 
 trafficState );

 break;
 }
}

Traffic Call Back Routine

This is the callback function supplied in the call to tdiTrafficRegister(). The Tserver calls this when the traffic feature has been disabled for the entire system or for this driverID.

void trafficCB ( TDIHandle_t driverID, int cbParam )
{
 int state;
 switch ( cbParam )
 { case TDI_TRAFFIC_ENABLED:
{ 
    state = cbParam;

    tdiLogError(DRIVER_NAME, 
                AUDIT_TRAIL, 
                LOC_CODE, 
                ERR_NO, 
                "Enabling traffic measurements" );
    break;
} 
}
case TDI_TRAFFIC_DISABLED:
{
    state = cbParam;

    tdiLogError(DRIVER_NAME, 
                AUDIT_TRAIL, 
                LOC_CODE, 
                ERR_NO, 
                "Disabling traffic measurements" );
    break;
}
default:
{
    state = getTrafficState (driverID);

    tdiLogError(DRIVER_NAME, 
                WARNING, 
                LOC_CODE, 
                ERR_NO, 
                "Tserver requested unsupported traffic 
                option %ld, traffic options will 
                NOT be changed", 
                cbParam );
    break;
}
}
setTrafficState ( state, driverID );
}

---

**Reporting Traffic**

This function calls the tdiTrafficReport function if the traffic state is enabled for this driverID.

```c
void reportTraffic ( 
    TDIHandle_t driverID, 
    TDIStatType_t statType, 
    TDIStatValue_t statValue )
{
    int rc;
    int trafficState;

    trafficState = getTrafficState (driverID);

    switch ( trafficState )
    {
    case TDI_TRAFFIC_ENABLED:
    {
        tdiLogError(DRIVER_NAME, 
                    TRACE, 
                    LOC_CODE, 
                    ERR_NO, 
```
"Reporting traffic type %ld value %ld for driverID %ld",
statType,
statValue,
driverID );
rc = tdiTrafficReport ( driverID,
   TDI_PBX_DRIVER,
   statType,
   statValue );

switch ( rc )
{
    case TDI_SUCCESS:
    {
        break;
    }
    case TDI_ERR_TRAFFIC_OFF:
    {
        /*
         * internal state was mis-set
         */
        trafficCB(driverID,
           TDI_TRAFFIC_DISABLED );
        break;
    }
    default:
    {
        tdiLogError(DRIVER_NAME,
           ERROR,
           LOC_CODE,
           ERR_NO,
           "Error %ld occurred while
           reporting traffic
           measurements",
           rc );
        break;
    }
    break;
}

case TDI_TRAFFIC_DISABLED:
{
    tdiLogError(DRIVER_NAME,
       TRACE,
       LOC_CODE,
       ERR_NO,
       "A type %ld value %ld traffic event
       occurred for driverID %ld, but traffic
       is disabled",
       statType,
       statValue,
       driverID );
    break;
}

case NOT_REGISTERED_WITH_TDI:
{
    tdiLogError(DRIVER_NAME,
       TRACE,
       LOC_CODE,
       ERR_NO,
       "Not registered with TDI",
       driverID );
    break;
}
ERR_NO,
"A type %ld value %ld traffic event
occurred for driverID %ld, but
this driver is not registered for
traffic measurements",
statType,
statValue,
driverID );
break;
}

default:
{
  tdiLogError(DRIVER_NAME,
  WARNING,
  LOC_CODE,
  ERR_NO,
  "A type %ld value %ld traffic event
occurred for driverID %ld, but
traffic reporting is in an
unknown state %ld",
  statType,
  statValue,
  driverID,
  TrafficState[driverID ];
  break;
}
}

Processing anacsOpenStream() Request

This section contains an example of handling the AcsOpenStream() request by
either returning an ACSOpenStreamConfEvent or an
ACSUniversalFailureConfEvent. This section illustrates how to set up the
Driver Control Block.

Receiving anACSOpenStreamConfEvent

Set a pointer to the buffer received from the Tserver to look at the DC block.

TDIDriverControlBlock_t *idc; /* incoming DC Block */
char *buffer; /* incoming buffer */

tdiReceiveFromTserver( tsdiHandle, &buffer);
idc = (TDIDriverControlBlock_t *) buffer;

Allocate a buffer large enough to hold the DC block, any private data, and the
confirmation event if one is being returned or the
ACSUniversalFailureConfEvent if the open request is being rejected. The
versions passed in the request should be verified. If any is not supported by the
driver, an ACSUniversalFailureConfEvent should be returned.

char *obuffer; /* outgoing buffer */
ACSOpenStream_t  *iMag; /* request msg */
```c
int rc;
TDIBuf_flag_t buf_flag;

/* alloc a buffer for the ACSOpenStreamConfEvent */
tdiAllocBuffer(tsdiHandle, &obuffer,
    (sizeof(TDIDriverControlBlock_t) +
     privateLength +
     sizeof(ACSOpenStreamConfEvent_t)),
    &buf_flag);

or

/* alloc a buffer for the ACSUniversalFailureConfEvent */
tdiAllocBuffer(tsdiHandle, &obuffer,
    (sizeof(TDIDriverControlBlock_t) +
     privateLength +
     sizeof(ACSUniversalFailureConfEvent_t)),
    &buf_flag);

/* set a ptr to the allocated buffer to fill in
* the confirmation event or universal failure event */
ACSOpenStreamConfEvent_t *oEvent; /* conf event */
oEvent = (ACSOpenStreamConfEvent_t *)
    (obuffer + sizeof(TDIDriverControlBlock_t) );

or

ACSUniversalFailureConfEvent_t *oEvent; /* failure event */
oEvent = (ACSUniversalFailureConfEvent_t *)
    (obuffer + sizeof(TDIDriverControlBlock_t) );

Set a pointer to the open request structure in the buffer received from the Tserver. The request message is located in the buffer specified by the messageOffset field in the DC block.

iMsg = (ACSOpenStream_t *)(buffer + idc->messageOffset);

---

**Returning an ACSOpenStreamConfEvent**

Fill in the parameters in the confirmation event. The driver version parameter must be set with the version of your driver and the remaining version fields should be returned as they were received.

ACSOpenStreamConfEvent_t *oEvent; /* conf event */
TDIDriverControlBlock_t *odc; /* outgoing DC Block */
iMsg = (ACSOpenStream_t *)(buffer + idc->messageOffset);
oEvent = (ACSOpenStreamConfEvent_t *)
    (obuffer + sizeof(TDIDriverControlBlock_t) );

/*
* Fill in the driver version and return
* the other version fields.
*/
strcpy(oEvent->drvrVer, yourDriverVersionString);
strcpy(oEvent->apiVer, iMsg->apiVer);
strcpy(oEvent->libVer, iMsg->libVer);
strcpy(oEvent->tsrvVer, iMsg->tsrvVer);

/*
```

---

**Issue 1 — December 2002**

---

4-10 TSDI.PDF — Avaya CT 1.2
* Fill in the DC Block for the return message.
*/

doctor = (TDIDriverControlBlock_t *) obuffer;

odc->messageOffset = sizeof(TDIDriverControlBlock_t);
odc->messageLength = sizeof(ACSOpenStreamConfEvent_t);
odc->privateOffset = odc->messageOffset + odc->messageLength;
odc->privateLength = privateLength;
odc->invokeID = idc->invokeID;
odc->sessionID = idc->sessionID;
odc->messageClass = ACSCONFIRMATION;
odc->messageType = ACS_OPEN_STREAM_CONF;
odc->class_of_service = idc->class_of_service;
odc->monitorCrossRefID = 0;

/* send the confirmation event to the Tserver */
rc = tdiSendToTserver(tsdiHandle, obuffer, TDI_NORMAL_MESSAGE);
if (rc != TDI_SUCCESS)
{
    tdiLogError(DRIVER_NAME, ERROR, LOC_CODE, ERR_NO,
        "tdiSendToTserver failed: rc = %d", rc);
}

---

Returning an
ACSUniversalFailureConfEvent

Fill in the error code for the failure.

TDIDriverControlBlock_t *odc; /* outgoing DC Block*/
ACSUniversalFailureConfEvent_t *oEvent; /* failure event */
oEvent = (ACSUniversalFailureConfEvent_t *)
    (obuffer + sizeof(TDIDriverControlBlock_t));

doctor = (TDIDriverControlBlock_t *) obuffer;

oEvent->error = yourErrorCode;

/*
* Fill in the DC Block for the return message.
*/

odc->messageOffset = sizeof(TDIDriverControlBlock_t);
odc->messageLength = sizeof(ACSUniversalFailureConfEvent_t);
odc->privateOffset = odc->messageOffset + odc->messageLength;
odc->privateLength = privateLength;
odc->invokeID = idc->invokeID;
odc->sessionID = idc->sessionID;
odc->messageClass = ACSCONFIRMATION;
odc->messageType = ACS_UNIVERSAL_FAILURE_CONF;
odc->class_of_service = idc->class_of_service;
odc->monitorCrossRefID = 0;

/* send the confirmation event to the Tserver */
rc = tdiSendToTserver(tsdiHandle, obuffer, TDI_NORMAL_MESSAGE);
if (rc != TDI_SUCCESS)
{
    tdiLogError(DRIVER_NAME, ERROR, LOC_CODE, ERR_NO,
        "tdiSendToTserver failed: rc = %d", rc);
}
## Processing an `acsCloseStream()` Request

```c
TDIDriverControlBlock_t *idc; /* incoming DC */
TDIDriverControlBlock_t *odc; /* incoming DC */
char *obuffer; /* outgoing buffer */
char *buffer; /* incoming buffer */
TDIBuf_flag_t buf_flag;

/* Set a pointer to the buffer received from
 * the Tserver to look at the DC Block. */

tdiReceiveFromTserver(tsdiHandle, &buffer);

idc = (TDIDriverControlBlock_t *) buffer;

/* Allocate a buffer large enough to hold the DC block,
 * any private data, and the confirmation event. */

idc = (TDIDriverControlBlock_t *) buffer;
tdiAllocBuffer(tsdiHandle, &buffer,
(sizeof(TDIDriverControlBlock_t) +
privateLength +
sizeof(ACSCloseStreamConfEvent_t)),
&buf_flag);

/*
 * Fill in the DC Block for the return message.
 */

odc = (TDIDriverControlBlock_t *) obuffer;

odc->messageOffset = sizeof(TDIDriverControlBlock_t);
odc->messageLength = sizeof(ACSCloseStreamConfEvent_t);
odc->privateOffset = odc->messageOffset + odc->messageLength;
odc->privateLength = privateLength;
odc->invokeID = idc->invokeID;
odc->sessionID = idc->sessionID;
odc->messageClass = ACS_CONFIRMATION;
odc->messageType = ACS_CLOSE_STREAM_CONF;
odc->class_of_service = idc->class_of_service;
odc->monitorCrossRefID = 0;

rc = tdiSendToTserver(tsdiHandle, obuffer, TDI_NORMAL_MESSAGE);
if (rc != TDI_SUCCESS)
{
    tdiLogError(DRIVER_NAME, ERROR, LOC_CODE, ERR_NO,
    "tdiSendToTserver failed: rc = %d", rc);
}
```

## Creating a `CSTAConferenceCallConfEvent`:

This section contains the structures (taken from cstadefs.h) needed to create a `CSTAConferenceCallConfEvent` and some hints on how to populate the buffer containing the confirmation event.

```c
typedef struct CSTAConferenceCallConfEvent_t {
    ConnectionID_t newCall;

    Issue 1 — December 2002
```
Allocate a buffer large enough to hold the DC block, any private data, the conference call confirmation event structure and, most importantly, the number of
connections supported on a conference multiplied by the size of the Connection structure.

The buffer will look as follows:

<table>
<thead>
<tr>
<th>ck</th>
<th>CSTA Conference Call ConfEvent</th>
<th>connection[0]</th>
<th>connection[1]</th>
<th>...</th>
<th>connection[count-1]</th>
<th>privateData</th>
</tr>
</thead>
</table>

The call to allocate the buffer may look like this:

```c
tdiAllocBuffer ( tdiHandle, 
    &buffer,  
    ( sizeof(TDIDriverControlBlock_t) + 
    privateLength + 
    sizeof(CSTAConferenceCallConfEvent_t) + 
    (NumberOfConnectionsSupported * 
    sizeof(Connection_t))), 
    &buf_flag);
```

Before populating the confirmation event, set a pointer to the location in the buffer after the DC block.

```c
CSTAConferenceCallConfEvent_t *ptr;  
ptr = (CSTAConferenceCallConfEvent_t *) 
    (buffer + sizeof(TDIDriverControlBlock_t));
```

Populate the newCall structure with the relevant information.

```c
ptr->newCall.callID = ... 
```

Set connList.count to the number of parties on the conference call.

```c
ptr->connList.count = NumberOfConnectionsOnTheCall;
```

Set the connList.connection pointer to the position in the buffer after the CSTAConferenceCallConfEvent by setting connList.connection to:

```c
ptr->connList.connection = (Connection_t *) 
    ((char *)ptr + sizeof(CSTAConferenceCallConfEvent_t));
```
Index into connList.connection and set the information for each connection.

```c
for( i=0; i < ptr->connList.count; i++ )
{
    ptr->connList.connection[i].party.callID = ...
}
```

### Private Data

There are two length fields to be set when sending and receiving private data. The first one is the `privateLength` field in the DC block. Remember that this field includes the size of the user-supplied data as well as the size of the `PrivateData_t` header. Therefore, the driver must expect to receive the size of the header in the `privateLength` field and must also include the size of this structure in the `privateLength` field when sending a message to the T server.

The second `length` field that must be set is the length field in the `PrivateData_t` header structure. This is the size of the user supplied data that is contained in the `data` field. The data field contains the first byte of the user's data. Thus, `privateLength` is set by subtracting one from the size of the header (since the data field in the header includes one byte of the user's data) and adding to it the size of the actual private data.

```c
/* defined in acs.h */
typedef struct PrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[1];
} PrivateData_t;
```

Private Data_t *privateData;

```c
privateData->length = sizeof(yourOwnDefinedPrivateDataStructure);
/* Fill in your private data */
memcpy(privateData->data, &yourOwnDefinedPrivateDataStructure);
/* Set the private data length in the DC block */
odc->privateLength = sizeof(PrivateData_t) - 1 +
    privateData->length;
```

### Processing a Monitor Request

This section illustrates setting up a confirmation event to a monitor device request followed by one of the unsolicited events that may result from a subsequent use of that device.

Perform any validation of the request. If the request is denied, return a `CSTAUniversalFailureConfEvent`. If the request is granted, return a...
CSTAMonitorConfEvent. Note that *ibuffer* is the buffer received from the Tserver containing the monitor device request.

```
TDIDriverControlBlock_t *odc, *idc;
char *obuffer; /* outgoing buffer */
CSTAMonitorConfEvent_t *oEvent;/* CSTAMonitorConfEvent */
CSTAMonitorCrossRefID_t crossRefID;
CSTAMonitorDevice_t *iMsg;

idc  = (TDIDriverControlBlock_t *) ibuffer;
iMsg  = (CSTAMonitorDevice_t *)(ibuffer + idc->messageOffset);

Allocate a buffer large enough to hold the confirmation event, any private data, and the DC block. This example shows the tdiAllocBuffer() call for the monitor confirmation event. The universal failure would be allocated the same way by using "sizeof(CSTAUniversalFailureConfEvent)"

```
rc = tdiAllocBuffer(tdiHandle, &obuffer,
      (sizeof(TDIDriverControlBlock_t) +
       privateLength +
       sizeof(CSTAMonitorConfEvent_t),
       &buf_flag);
```

```
/* set the fields in the monitor confirmation event */
oEvent = (CSTAMonitorConfEvent_t *)
      (obuffer + sizeof(TDIDriverControlBlock_t ));

    oEvent->monitorCrossRefID = ADriverCrossRefID;
oEvent->monitorFilter.call = x;
oEvent->monitorFilter.feature = x;
oEvent->monitorFilter.agent = x;
oEvent->monitorFilter.maintenance = x;
oEvent->monitorFilter.privateFilter = x;

    /* Fill in the DC Block for the return message. */
    odc  = (TDIDriverControlBlock_t *) obuffer;

    odc->messageOffset = sizeof(TDIDriverControlBlock_t);
    odc->messageLength = sizeof(CSTAMonitorConfEvent_t);
    odc->privateOffset = odc->messageOffset + odc->messageLength;
    odc->privateLength = privateLength;
    odc->invokeID = idc->invokeID;
    odc->sessionID = idc->sessionID;
    odc->messageClass = CSTACONFIRMATION;
    odc->messageType = CSTA_MONITOR_CONF;
    odc->class_of_service = idc->class_of_service;
    odc->monitorCrossRefID = oEvent->monitorCrossRefID;
```

One of the unsolicited events that may be returned due to a monitor device request is the CSTA_DELIVERED event.

```
/* send the confirmation event to the Tserver */
* using tdiSendToTserver.
*/
```

/* allocate a buffer large enough to... */

Issue 1 — December 2002
* hold the DC block, any private data
* and the unsolicited event.
*/

CSTA*msg;
msg = (CSTA*obuffer + sizeof(TDIDriverControlBlock_t));

/* set the fields in CstaDeliveredEvent structure */
msg->connection = x;
msg->alertingDevice = x;
msg->callingDevice = x;
msg->calledDevice = x;
msg->lastRedirectionDevice = x;
msg->localConnectionInfo = x;
msg->cause = x;

/*
* Fill in the DC Block for the return message.
*/

odc->messageOffset = sizeof(TDIDriverControlBlock_t);
odc->messageLength = sizeof(CSTA);
odc->privateOffset = odc->messageOffset + odc->messageLength;
odc->privateLength = privateLength;
odc->invokeID = idc->invokeID;
odc->sessionID = idc->sessionID;
odc->messageClass = CSTAUNSOLICITED;
odc->messageType = CSTA_DELIVERED;
odc->class_of_service = idc->class_of_service;
odc->monitorCrossRefID = theMonitorCrossRefIDForthisMonitor;

/* send the unsolicited event to the Tserver
* using tdiSendToTserver.
*/
The following manual pages describe the function call interface between the Driver and the Tserver.

### tdiAllocBuffer()

This function is issued by the Tserver or the Driver to allocate a buffer for sending a message across the TSDI.

**Syntax**

```c
#include <tdi.h>

TDIReturn_t tdiAllocBuffer (
    TDIHandle_t driverID, /* INPUT */
    char **bufptr, /* OUTPUT */
    unsigned lint length, /* INPUT */
    TDIBuf_flag_t *buf_flag); /* OUTPUT */
```

**Parameters**

- **driverID**
  
  This is the value of the handle returned by the `tdiDriverRegister()` function call. This handle uniquely identifies the interface between the Tserver and the Driver.

- **bufptr**
  
  This parameter is set to point to the start of the buffer returned by the `tdiAllocBuffer()` function call. If the `tdiAllocBuffer()` call is not successful, the function will set `bufptr` to NULL. A buffer pointed to by `bufptr` is guaranteed to point to a byte aligned block of data.

- **length**
  
  This parameter specifies the size (in bytes) of the memory block. The `length` must be less than TDI_MAX_BUFFER_SIZE. If the `tdiAllocBuffer()`
routine is successful, this function will return a block of data that is at least
length bytes long. (The TSDI will allocate a block of memory that is larger than
length. The first 16 bytes of this memory block will be used by the TSDI to
implement the message queues; however, these 16 bytes should not be
included in the length field. The TSDI will automatically add the 16 bytes onto
the length field and create and maintain the header). This routine will return
bufptr as the first byte aligned point in the memory block after the message
header.

buf_flag

This parameter is a bit mask set by the tdiAllocBuffer() routine to provide
information on the amount of memory allocated by the Tserver or the Driver for
this interface. The buf_flag parameter will be set to indicate the following
conditions when they occur:

<table>
<thead>
<tr>
<th>TDI_EXCEED_HIWATER_MARK</th>
<th>The current amount of memory allocated for message buffers (by the TServer and the Driver) is greater than the high water mark (hiwater_mark) specified in the tdiDriverRegister() function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI_EXCEED_MAX_BYTES</td>
<td>The current amount of memory allocated for message buffers (by the TServer and the Driver) is greater than the maximum number of bytes (max_bytes) specified in the tdiDriverRegister() function. This will only be returned on a failure.</td>
</tr>
</tbody>
</table>

Return Values

This function returns TDI_SUCCESS on success, and bufptr is set to NULL.
This function returns one of the following negative values on failure:

<table>
<thead>
<tr>
<th>TDI_ERR_BAD_DRVRID</th>
<th>The current amount of memory allocated for This error indicates that the driverID specified in the tdiAllocBuffer() function is not valid.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI_ERR_NO_MEM</td>
<td>This error indicates the TSDI was unable to allocate the requested memory from the operating system.</td>
</tr>
<tr>
<td>TDI_ERR_BAD_LENGTH</td>
<td>This error indicates that the requested length is greater than TDI_MAX_BUFFER_SIZE.</td>
</tr>
<tr>
<td>TDI_ERR_NO_BUFFERS</td>
<td>This error indicates that the Tserver and the Driver have (together) allocated more memory for message buffers than allowed for this TSDI. The maximum amount of memory allowed is set via the max_bytes field of the buffer_descriptor parameter when the Driver registers with the TSDI.</td>
</tr>
</tbody>
</table>
The `tdiAllocBuffer()` function provides a buffer to the Tserver or the Driver that can be used to send a message across the TSDI. The buffers are allocated from operating system if the current amount of memory allocated for this interface is less than the maximum specified at Driver registration time. The Driver is responsible for setting the maximum bytes allowed for this interface during the `tdiDriverRegister()` routine. If the Driver has allocated a message buffer via the `tdiAllocBuffer()` routine, the Driver is responsible for dealing with the buffer by either sending the buffer back to the Tserver through the TSDI via the `tdiSendToTserver()` function, in which case the Tserver is responsible for freeing the buffer via the `tdiFreeBuffer()` function. If the Tserver has allocated a message buffer via the `tdiAllocBuffer()` routine, the Tserver is responsible for freeing the buffer back to the TSDI either by successfully sending the buffer to the Driver via the `tdiSendToDevice()` function, or freeing the buffer via the `tdiFreeBuffer()` function.

The `tdiAllocBuffer()` function will return a bit mask, `buf_flag`, indicating the current status of the memory allocated for message buffers used on this interface. The Tserver and the Driver can examine this bit mask to determine if some form of voluntary flow control is required.

Memory allocated for message buffers via the `tdiAllocBuffer()` routine should not be directly freed back to the operating system by the Tserver or the Driver; the messages should be released back to the TSDI via the `tdiFreeBuffer()` routine.

**WARNING:**

The Tserver and Driver are not guaranteed to receive a message buffer via the `tdiAllocBuffer()` routine even though the current memory allocated is less than `max_bytes` specified by the Driver in the `tdiDriverRegister()` routine. The operating system may not have the resources at this time to fulfill the memory allocation request.

**Driver Notes**

The Driver is responsible for issuing a `tdiDriverRegister()` function call to specify the maximum number of bytes that can be allocated for message buffers by the Tserver or the Driver for this interface. The `driverID` returned by the `tdiDriverRegister()` routine must be used to allocate buffers that will be used across this interface. The Driver should monitor the `buf_flag` parameter to determine if the memory allocation limit is sized appropriately and to determine when some form of flow control is required. The Driver is not responsible for the memory resources allocated by `tdiAllocBuffer()` routine since they are

TSDI Manual Pages

Issue 1 — December 2002
allocated from the operating system by the TSDI. The TSDI is responsible for freeing these memory resources. The Driver is responsible for giving the memory resources back to the TSDI via the tdiFreeBuffer() routine, or sending the buffer to the Tserver via the tdiSendToTserver() routine.

Tserver Notes

The driverID must be used to allocate buffers that will be used across this TSDI. The Tserver is responsible for giving the memory resources back to the TSDI via the tdiFreeBuffer() routine, or sending the buffer to the Driver via the tdiSendToDriver() routine. The TSDI is part of the Tserver, and the memory allocated for the TSDI must be freed before the Tserver can unload.

tdiDriverRegister()

This function allows a Driver (PBX or other) to register a service with the Tserver. The Driver registers its name with the Tserver, specifies some tags that will be maintained by the Tserver for maintenance queries, and requests parameters that specify the memory allocation limits that will be imposed on the Tserver and Driver for this interface. The Driver also indicates which version of the TSAPI protocol this driver can talk over ACS streams established for this Driver service. The protocol descriptor is a new argument to this routine. Actually, the argument replaces the channel_number argument that was defined in Release 1. Both the protocol descriptor and channel_number arguments occupy the same space in the argument list. This simplifies the matter of compatibility between old and new Tservers and Driver. The Release 2 Tserver will continue to support both methods of calling this function, that is, with TSAPI protocol descriptor argument or with the channel_number argument.

Syntax

```
#include <tdi.h>

TDIHandle_t tdiDriverRegister ( 
    const char *service_name,    /* INPUT */
    const char *driver_name,     /* INPUT */
    int service_type,            /* INPUT */
    TDIProtocol_t protocol_descriptor,    /* INPUT */
    const char *vendor_name,     /* INPUT */
    const TDIVersion_t version,   /* INPUT */
    TDISecurity_t driver_security, /* INPUT */
    TDIBuf_info_t *buffer_descriptor);/* INPUT */
```

Parameters

- **service_name**
This is the NULL-terminated ASCII string that the Driver will provide to the Tserver to be used for service advertising. This parameter is mandatory and cannot be NULL (neither NULL pointer nor NULL string). It also must not exceed the maximum length of TDI_MAX_SERVICE_NAME. The service_name must be unique for every registration done by a Driver (for example, for every unique driver_name, the service_name must be unique).

**driver_name**

This is the NULL-terminated ASCII string that the Driver will provide to the Tserver to identify the Driver for OA&M and debugging purposes. This parameter is mandatory and cannot be NULL (neither NULL pointer nor NULL string). It also must not exceed the maximum length of TDI_MAX_DRIVER_NAME. A Driver may register more than once using the same driver_name. A Driver may also register more than once using different driver_names.

**service_type**

This parameter identifies the service class that will be advertised for the service_name. This parameter should be set to TDI_ST_CSTA for CSTA Services or TDI_ST_OAM for OA&M Services.

**protocol_descriptor**

This field replaces the unused channel_number field from Release 1. It is only used if the version field is equal to TSDI_VERSION_2. This field is a bit field containing information about the TSAPI protocol versions that this driver can support. The protocol versions are relative to the service_type; that is, the protocol version specifies either the versions of TSAPI OAM or TSAPI CSTA protocol. The header file tdi.h contains #defines for the various versions of TSAPI protocols that can be bit-wise OR-ed together. If the version field is TSDI_VERSION then this argument is ignored (this occurs in Release 1 Tservers), and the Release 1 protocol versions are assumed.

**vendor_name**

This is the NULL-terminated ASCII string that identifies the manufacturer’s name of the Driver. This parameter is mandatory and cannot be NULL (neither NULL pointer nor NULL string). It also must not exceed the maximum length of TDI_MAX_VENDOR_NAME.

**version**

This parameter is set to the version of the TSDI with which the Driver will function and is mandatory. The registration will fail if this parameter is set to an invalid version. This field is a pointer to a string that contains the version. Valid versions are TSDI_VERSION for Release 1 and TSDI_VERSION_2 for Release 2.

**driver_security**
This parameter indicates whether or not the Driver wants the Tserver to provide security checks from its security database for the CSTA nonprivate portion of each message from the client. This parameter is mandatory and must be set to one of the following:

- **TDI_CSTA_SECURITY**
  
  Login and Password will be validated on the `acsOpenStream()` request.
  
  Entry in the Tserver's Security Database must also contain this login.
  
  Each subsequent CSTA request will be validated per the user's administered permissions.

- **TDI_LOGIN_SECURITY**
  
  Login and Password will be validated on the `acsOpenStream()` request.
  
  Entry in the Tserver's Security Database must also contain this login.

- **TDI_NO_SECURITY**
  
  No validation is done on any CSTA or ACS messages.

If the Telephony Services Security database (SDB) is disabled by administration, and if the driver registers with `driver_security TDI_CSTA_SECURITY`, the Avaya CT Server will not do CSTA device-level permission validation because no permissions are available; hence the effective registration will be `TDI_LOGIN_SECURITY`.

**buffer_descriptor**

This is a pointer to a buffer descriptor structure containing the information about the memory that can be allocated for this TSDI. A NULL pointer will use the default values listed under the field definitions section for each element of the buffer descriptor.

```c
typedef struct
{
    unsigned long   max_bytes; /* Maximum number of bytes to allocate for this interface */
    unsigned long   hiwater_mark; /* High-water mark for buffer allocation on this interface */
} TDIBuf_info_t;
```

**NOTE:** The values of `max_bytes` and `hiwater_mark` are stored in the Tserver Security Database and are used in place of the values provided on the register for all subsequent loads of the driver. These values can also be changed via the Tserver's OA&M application.

**Return Values**
This function returns a *driverID* on success that must be used in all subsequent function calls by the Tserver and the Driver to identify this specific Tserver-Driver interface. The *driverID* is guaranteed to be a positive integer. On failure, this function returns one of the following negative values:

- **TDI_ERR_DUP_DRVR**
  This error indicates that the combination of *vendor_name*, *service_name*, and *service_type* provided has already been registered with the Tserver.

- **TDI_ERR_MAX_DRVR**
  This error indicates that the maximum number of registered drivers, TDI_MAX_REGISTRATIONS, has been reached.

- **TDI_ERR_EINVAL**
  This error indicates that an invalid parameter was specified for the tdiDriverRegister() function call.

- **TDI_ERR_BAD_VERSION**
  This error indicates that an invalid version number was supplied in the version parameter.

- **TDI_ERR_BAD_PROTOCOL**
  This error indicates that the Tserver does not understand any of the protocol versions specified.

- **TDI_ERR_ESYS**
  This error indicates that some form of system error has occurred. When this occurs, the TSDI will place an entry in the Error Log.

### Comments

This function is issued by the Driver to set up a communication path with the Tserver; identify the name that will be advertised by the Tserver (the name is generated from the *vendor_name*, *service_name*, and *service_type* parameters); and specify the maximum amount of memory that may be used for message buffers used to exchange messages between the Tserver and the Driver for this communication path. Both the Tserver and the Driver must allocate message buffers from the TSDI routines to send a message across this TSDI.

When a Driver registers with the TSDI, it must specify the maximum amount of memory that can be allocated for message buffers by the Driver and the Tserver for this communication path. Each message buffer allocated by the Driver or the Tserver from the TSDI will include a (16 byte) header that will be used to implement the monitoring and queuing of messages. *(The message buffer header should not be accessed by the Driver or the Tserver, it is only intended*
for use by the TSDI routines.) This header is not charged to the space allocated in the TSDI via the max_bytes field defined below.

The structure of type TDIBuf_info_t is defined as follows:

```c
typedef struct
{
    unsigned long max_bytes,
    unsigned long hiwater_mark,
} TDIBuf_info_t;
```

Field definitions:

**max_bytes**

A non-negative integer indicates the maximum amount of memory that can be allocated by the T server and the Driver for message buffers used on this communication path between the Driver and T server. The tdiAllocBuffer() routine will fail all requests when the amount of memory currently allocated for this interface exceeds max_bytes. If the buffer_descriptor parameter is a NULL pointer, this value will default to TDI_MAX_BYTES_ALLOCATED.

**hiwater_mark**

A non-negative integer indicates a high water mark for the memory allocated for this TSDI. When the amount of memory allocated for this interface exceeds the high-water mark, the tdiAllocBuffer() routine will return the buffer to the caller (if a memory block can be allocated from the operating system), and indicate that the high-water mark has been exceeded. If the buffer_descriptor parameter is a NULL pointer, this value will default to TDI_BUFFER_HI_WATER_MARK.

### tdiDriverSanity()

The Driver calls this function once a minute to report to the T server that it is alive and functioning. If the T server does not receive this message, it will place an ERROR in the error logging file and send this error condition message to the T server's OA&M client only if the client application is up and running. If the T server's OA&M client application is not running at the time the T server detects the Driver has not indicated its sanity, no message is sent (and will not be sent even if the T server's OA&M application is started at a later time).

### Syntax

```c
#include <tdi.h>

TDIReturn_t tdiDriverSanity(
```

Issue 1 — December 2002

**TSDI.PDF — Avaya CT 1.2**
TDIHandle_t driverID) /* INPUT */

Parameters

**driverID**

This is the unique identification number given to the Driver when it registered with the T server via `tdiDriverRegister()`. It allows the routine to identify correctly which buffer queue to use to access the message from the T server.

Return Values

This function returns **TDI_SUCCESS** on success, and returns one of the following negative values on failure:

| TDI_ERR_BAD_DRVRIID | This error indicates that the **driverID** specified in the `tdiDriverSanity()` function is not valid. |

Driver Notes

The Driver is responsible for calling this function once every minute.

**tdiDriverUnregister()**

This function allows a Driver (PBX or other) to unregister itself with the T server. It must use the driver_id that was returned by the `tdiDriverRegister()` routine.

Syntax

```c
#include <tdi.h>

TDIReturn_t tdiDriverUnregister ( 
    TDIHandle_t driverID); /* INPUT */
```

Parameters

**driverID**
This is the unique identification number given to the Driver when it registered with the Tserver.

Return Values

This function returns TDI_SUCCESS on success, and returns one of the following negative values on failure

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI_ERR_BAD_DrvRID</td>
<td>This error indicates that the driverID specified in the tdiDriverUnregister() function is not valid.</td>
</tr>
<tr>
<td>TDI_ERR_ESYS</td>
<td>This error indicates that some form of system error has occurred. When this occurs, the TSDI will place an entry in the Error Log.</td>
</tr>
</tbody>
</table>

Comments

This routine is exported from the Tserver for use by Drivers that import the TSDI routines. It will cause the Tserver to delete any resources created when the driver originally registered with the Tserver via the tdiDriverRegister() routine. All memory that was allocated for this TSDI will be given back to the operating system. All messages in the queues will be removed and the queues will be deleted. If the driver unloads before calling this routine, the Tserver will attempt to deallocate all of the resources associated with this TSDI.

tdiFreeBuffer ()

This function is issued by the Tserver or Driver to free a buffer that was previously allocated to transmit a message across the TSDI.

Syntax

```c
#include <tdi.h>

TDIReturn_t tdiFreeBuffer ( 
    TDIHandle_t driverID, /* INPUT */
    char *bufptr); /* INPUT*/
```

Parameters
driverID

This is the value of the handle returned by the tdiDriverRegister() function call. This handle uniquely identifies the TSDI.

bufptr

This parameter is a pointer to the start of the buffer returned by the tdiAllocBuffer() function call for either the Tserver or the Driver, a tdiReceiveFromDriver() function call for the Tserver, or a tdiReceiveFromTserver() function call for the Driver. After the tdiFreeBuffer() routine completes, the caller should no longer access the buffer.

Return Values

This function returns TDI_SUCCESS on success, and returns one of the following negative values on failure:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI_ERR_BAD_DRVRID</td>
<td>This error indicates that the driverID specified in the tdiFreeBuffer() function is not valid.</td>
</tr>
<tr>
<td>TDI_ERR_BAD_BUF</td>
<td>This error indicates that the memory block pointed to by bufptr is not a currently allocated TSDI buffer.</td>
</tr>
<tr>
<td>TDI_ERR_NOT_YOUR_BUFFER</td>
<td>This error indicates that the driverID specified did not match the driverID stored with this TSDI Buffer when the buffer was created via the tdiAllocBuffer() call. A Driver is only allowed to free TSDI Buffers that was originally allocated for this Driver. <strong>NOTE:</strong> All TSDI Buffers allocated by a Tserver to be sent across the TSDI to a Driver are allocated with that Driver's ID. When this occurs, the TSDI will place an entry in the Error Log.</td>
</tr>
<tr>
<td>TDI_ERR_ESYS</td>
<td>This error indicates that some form of system error has occurred. When this occurs, the TSDI will place an entry in the Error Log.</td>
</tr>
</tbody>
</table>

Comments
The `tdiFreeBuffer()` function returns a buffer to the operating system that was previously allocated to send a message between the Driver and the Tserver.

⚠️ **WARNING:**

Memory allocated for message buffers via the `tdiAllocBuffer()` routine should not be directly freed back to the operating system by the Tserver or the Driver. The messages should be released back to the TSDI via the `tdiFreeBuffer()` routine.

Memory allocated from OS directly may not be released back to the OS by the `tdiFreeBuffer()` routine.

### Driver Notes

The Driver is responsible for issuing a `tdiDriverRegister()` function call to specify the maximum number of bytes that can be allocated for message buffers by the Tserver or the Driver for this interface. The `driverID` returned by the `tdiDriverRegister()` routine must be used to free buffers that have been allocated from the TSDI. The Driver is not responsible for the memory resources allocated by the TSDI since they are allocated from the operating system. The TSDI is responsible for freeing these memory resources. The Driver is responsible for giving the memory resources back to the TSDI via the `tdiFreeBuffer()` routine or sending the buffer to the Tserver via the `tdiSendToTserver()` routine.

### Tserver Notes

The `driverID` must be used to free buffers that have been allocated from this TSDI. The Tserver is responsible for giving the memory resources back to the TSDI via the `tdiFreeBuffer()` routine after a message has been processed, or sending the message to the Driver via the `tdiSendToDriver()` routine. The TSDI is part of the Tserver, and the memory allocated for the TSDI must be released before the Tserver can unload.

### `tdiGetSessionIDInfo()`

This function will return the current information related to an ACS session. This function assumes SPX/IPX as the transport provider and is provided for backward compatibility only. For TSDI version 2, the function `tdiGetSessionInformation()` should be used instead of this function.

#### Syntax

```c
#include <tdi.h>

TDIReturn_t tdiGetSessionIDInfo ( 
    TDIHandle_t driverID,  /* INPUT */ 
    /* ... */
```

Issue 1 — December 2002
SessionID_t  sessionID, /* INPUT */
TDISessionID_t  *sessionIDInfo); /* OUTPUT */

Parameters

**driverID**

This is the unique identification number given to the driver when it registered with
the Tserver via the tdiDriverRegister() routine.

**sessionID**

This is the ACS sessionID assigned to this session by the Tserver when the
ACSOpenStreamRequest was sent to the driver.

**sessionIDInfo**

This parameter is a pointer to a TDISessionID_t structure. The Tserver will fill
information related to this session in the structure pointed to by this parameter.

typedef struct
LoginID_t   loginID;   /* Login for this session */
AppName_t   appName;  /* Application name for session*/
unsigned long network; /* Network of worktop */
unsigned char node[6]; /* Node of worktop */
LoginTime_t  timeOpened; /* Time the ACS stream opened */
char         homeDeviceID[16];/* Primary device ID of Home
   */
char         awayDeviceID[16];/* Primary device ID of
   */
} TDISessionID_t;

Return Values

This function returns **TDI_SUCCESS** on success, and this function sets the
return parameters to 0 and returns one of the following negative values on
failure:

- **TDI_ERR_BAD_DRVRLID** This error indicates that the **driverID** specified in the tdiGetSessionIDInfo() function is not valid.
- **TDI_ERR_BAD_SESSIONID** This error indicates that the **sessionID** specified in the tdiGetSessionIDInfo() function is not valid.

Comments

Issue 1 — December 2002
This routine is exported from the Tserver for use by the Drivers, and can be called internally by the Tserver. It provides a method to obtain information about an ACS session that would be useful in debugging or generating trace statements from the driver.

This function assumes SPX/IPX as the transport provider. If the session for which information is requested is not associated with an SPX/IPX transport connection, then the network and node fields of the returned structure will be set to zero.

**tdiGetSessionInformation()**

This function will return the current information related to an ACS session. Unlike tdiGetSessionIDInfo(), this function can return either the SPX/IPX or TCP/IP address of the client associated with the session. This function is only available in Release 2 and later versions of the Tserver, and replaces the tdiGetSessionIDInfo() function.

**Syntax**

```c
#include <tdi.h>

TDIReturn_t tdiGetSessionInformation(
    TDIHandle_t driverID, /* INPUT */
    SessionID_t sessionID, /* INPUT */
    TDISessionInfo_t *sessionIDInfo); /* OUTPUT */
```

**Parameters**

- **driverID**
  This is the unique identification number given to the driver when it registered with the Tserver via the tdiDriverRegister() routine.

- **sessionID**
  This is the ACS sessionID assigned to this session by the Tserver when the ACSOpenStreamRequest was sent to the driver.

- **sessionIDInfo**
This parameter is a pointer to a TDISessionInfo_t structure. The Tserver will fill information related to this session in the structure pointed to by this parameter.

typedef struct
{
    unsigned long network; /* Network of worktop */
    unsigned char node[6]; /* Node of worktop */
} TDISPXAddr_t;

typedef struct
{
    unsigned long addr; /* 32 byte address */
} TDIIPAddr_t;

typedef struct
{
    LoginID_t login; /* Login for */
    /* this session */
    AppName_t appName; /* App name for */
    /* this session */
    LoginTime_t timeOpened; /* Time stream */
    /* was opened */
    char homeDeviceID[16]; /* Primary deviceID of */
    /* Home Worktop */
    /* record */
    char awayDeviceID[16]; /* Primary deviceID of */
    /* Away Worktop */
    /* record */
    unsigned short transportType; /* Transport type */
    union {
        char pad[32]; /* struct padding */
        TDISPXAddr_t spxAddress; /* SPX/IPX address */
        TDIIPAddr_t ipAddress; /* IP address */
    } address;
} TDISessionInfo_t;

/* defines for transportType field */
#define TDI_SPX_IPX1
#define TDI_TCP_IP2

Return Values
This function returns **TDI_SUCCESS** on success, and sets the return parameters to 0 and returns one of the following negative values on failure:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI_ERR_BAD_DRV RID</td>
<td>This error indicates that the <em>driverID</em> specified in the <em>tdiGetSessionInformation()</em> function is not valid.</td>
</tr>
<tr>
<td>TDI_ERR_BAD_SESSIONID</td>
<td>This error indicates that the <em>sessionID</em> specified in the <em>tdiGetSessionInformation()</em> function is not valid.</td>
</tr>
</tbody>
</table>

**Comments**

This routine is exported from the Tserver for use by Drivers, and can be called internally by the Tserver. It provides a method to obtain information about an ACS session that would be useful in debugging or generating trace statements from the Driver.

### tdiGetTDISize()

This function will return the current administered size of the TSDI for the indicated driver. The TSDI size and high-water mark is initially set by the driver in the *tdiDriverRegister()* routine. The TSDI size, however, can be changed via the Tserver’s OA&M application. This routine returns the current size and high-water mark of the TSDI.

**Syntax**

```c
#include <tdi.h>

TDIReturn_t tdiGetTDISize(
    TDIRaw_t driverID, /* INPUT */
    TDIRaw_t *buffer_descriptor)/* OUTPUT */
```

**Parameters**

*driverID*

This is the unique identification number given to the driver when it registered with the Tserver via the *tdiDriverRegister()* routine.
buffer_descriptor

This is a pointer to a buffer_descriptor structure that will be returned by the Tserver. The buffer_descriptor structure has the following format:

typedef struct
{
    unsigned long max_bytes; /* Maximum number of bytes to
        * allocate for this interface
    */
    unsigned long hiwater_mark; /* High water mark for buffer
        * allocation
    */
} TDIBuf_info_t;

Return Values

This function returns TDI_SUCCESS on success, and sets the return parameters to 0 on failure and returns the following negative value:

TDI_ERR_BAD_DRVRID This error indicates that the driverID specified in the tdiGetTDISize() function is not valid.

tdiGetTservVersion()

This function will get the Customer and Internal version numbers of the Tserver. It also returns the highest available TSDI version and the supported TSAPI protocol versions for a given service type.

Syntax

#include <tdi.h>

TDIReturn_t tdiGetTservVersion(
    char *customer_version, /* OUTPUT */
    char *internal_version, /* OUTPUT */
    TDIVersion_t tsdi_version,  /* OUTPUT */
    int service_type, /* INPUT */
    TDIProtocol_t *protocol_version) /* OUTPUT */

Parameters
**customer_version**

This is the customer version of the Telephony Server (for example, 2.1a).

**internal_version**

This is the internal version of the Telephony Server (for example, 2.1a.1.0)

**tsdi_version**

This is the highest version of the TSDI interface supported (for example, 2.1).

**service_type**

This indicates the service type or stream type, TDI_ST_CSTA for CSTA Services or TDI_ST_OAM for OA&M Service, for which protocol version information is being requested.

**protocol_version**

This indicates the TSAPI protocol versions (CSTA or OAM based on the service_type field) that the Tserver supports. The returned value is a bit-field with bit 0 set if the version 1 protocol is supported, bit 1 set if the version 2 protocol is supported, and so forth. See the #defines in the header file tdi.h for a list of available protocol versions.

### Return Values

This function returns **TDI_SUCCESS** on success, and returns the following negative value on failure:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI_ERR_BAD_SRVC_TYPE</td>
<td>This error indicates that the service_type passed to the tdiGetTservVersion() function is not valid.</td>
</tr>
</tbody>
</table>

### Comments

A value of NULL may be passed in any output parameter to tdiGetTservVersion() if the associated information is not of interest to the caller. The maximum length of a returned customer_version, internal_version, or tsdi_version string is TDI_MAX_VERSION_STRING, not including the terminating NULL.
tdiLogError()

This function provides access to the common error logging mechanism provided by the Tserver. The Tserver and Driver use this function to log errors to a common log file, to the NT Event Log, or to send alarm messages to the Tserver OA&M application. These three destinations may be enabled or disabled for each error level through the Tserver's OA&M application.

Syntax

```c
#include <tdi.h>

void
tdiLogError(
    char *module, /* INPUT */
    ElogLevel_t level, /* INPUT */
    int location_code, /* INPUT */
    int error_code, /* INPUT */
    char *format, /* INPUT */
    ... /* INPUT */
)
```

Parameters

**module**

This is any null terminated string the Driver wishes to use to identify itself in the error log entries. It is recommended that this string match the name of the Driver.

**level**

This parameter indicates the level of the error and may be one of the following:

- **TRACE** trace message (for debugging transient problems)
- **CAUTION** unexpected software condition that is not fatal
- **AUDIT_TRAIL** important (normal) events: driver loaded, link reset, etc.
By default, only **CAUTION, AUDIT_TRAIL, WARNING, ERROR, and FATAL** errors are logged, but logging of lower level events can be enabled via the Tserver's OA&M application.

**location_code**

This parameter provides an indication of the location in the source code where this `tdiLogError()` request was made.

**error_code**

This parameter provides a concise way to identify specific error messages.

**format**

This parameter provides a `printf()-like` format string that uses a variable number (0 or more) of parameters that follow. No newline characters should appear in format or in any string or character arguments that follow `format`.

**Return Values**

This function returns nothing.

**Comments**

The `tdiLogError()` function provides a timestamp, date and time with 1 second resolution, and the format of the message is one line with date, time, module, location_code, error_code, level and the message generated by `format` and the optional parameters.

The actual logging of the message is done asynchronously by another thread, so the `tdiLogError()` function is non-blocking and returns control to the caller.

**WARNING**

a problem that of itself is not service-affecting, but indicates a condition that may become a problem (for example, low resources)

**ERROR**

a service-affecting condition that is not fatal

**FATAL**

a fatal problem to the logging module
very quickly. There is a limit to the number of messages allowed on this thread's queue at one time. If this backlog reaches 500, subsequent tdiLogError() requests will be ignored. An AUDIT_TRAIL error message will be generated when this occurs and placed in the error log file. It will also be sent to any other destination set for this level. This mechanism is in place to prevent the queue of messages from using too much memory.

Driver Notes

This function is intended for logging errors throughout the lifetime of the Driver (as well as the Tserver). tdiLogError() may be called prior to any tdiDriverRegister() and may be called after all tdiDriverUnregister() calls have been done.

tdiMapInvokeID()

This function will map the Tserver generated invokeID back into the invokeID generated by the client application.

Syntax

#include <tdi.h>

TDIReturn_t tdiMapInvokeID(
    TDIHandle_t driverID, /* INPUT */
    SessionID_t sessionID, /* INPUT */
    InvokeID_t invokeID, /* INPUT */
    InvokeID_t *appInvokeID); /* OUTPUT */

Parameters

   driverID

   This is the unique identification number given to the driver when it registered with the Tserver via the tdiDriverRegister() routine.

   sessionId

Issue 1 — December 2002
This is the ACS sessionID assigned to this session by the T server when the ACSOpenStreamRequest was sent to the driver.

**invokeID**

This parameter is the invokeID passed to the driver in the Driver Control block with a specific message.

**appinvokeID**

This parameter is the invokeID generated by the client application.

**Return Values**

This function returns **TDI_SUCCESS** on success, and sets the return parameters to 0 and returns one of the following negative values on failure:

- **TDI_ERR_BAD_DVRID**
  - This error indicates that the *driverID* specified in the tdiMapInvokeID() function is not valid.

- **TDI_ERR_BAD_SESSIONID**
  - This error indicates that the *sessionID* specified in the tdiMapInvokeID() function is not valid.

**Comments**

This routine is exported from the T server for use by Drivers and can be called internally by the T server. This routine provides a method of obtaining information about an ACS session that would be useful in debugging or generating trace statements from the driver.

**tdiMemAllocSize()**

This function will return the current status of the memory used for message buffers by a Tserver-Driver interface. This routine returns a count of the bytes in message buffers that are in each of the four possible states: queued to the Tserver, queued to the Driver, owned by the Driver, or owned by the Tserver.

Issue 1 — December 2002
Syntax

#include <tdi.h>

TDIReturn_t tdiMemAllocSize (
TDIReference_t driverID, /* INPUT */
TDIMemAlloc_info_t *mem_descriptor); /* OUTPUT */

Parameters

driverID
This is the unique identification number given to the driver when it registered with
the Tserver via the tdiDriverRegister() routine.

mem_descriptor
This parameter returns the following information in the TDIMemAlloc_info_t
structure:

typedef struct {
    unsigned long bytes_queued_to_driver;
    unsigned long bytes_queued_to_tserver;
    unsigned long bytes_allocd_by_driver;
    unsigned long bytes_allocd_by_tserver;
}TDIMemAlloc_info_t;

Where the sum of these fields is the total number of bytes allocated for message
buffers for this Tserver-Driver interface. The fields of TDIMemAlloc_info_t
are defined as follows:

bytes_queued_to_driver
This parameter specifies the number of bytes in message buffers that are
currently queued to the Driver.

bytes_queued_to_tserver
This parameter specifies the number of bytes in message buffers currently queued to the Tserver.

**bytes_allocd_by_driver**

This parameter specifies the number of bytes in message buffers currently allocated to the Driver. Message buffers are allocated to the Driver if it has performed a `tdiAllocBuffer()` or a `tdiReceiveFromTserver()` function call.

**bytes_allocd_by_tserver**

This parameter specifies the number of bytes in message buffers currently allocated to the Tserver. Message buffers are allocated to the Tserver if it has performed a `tdiAllocBuffer()` or a `tdiReceiveFromDriver()` function call.

**Return Values**

This function returns **TDI_SUCCESS** on success, and sets the return parameters to 0 on failure and returns one of the following negative values:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI_ERR_BAD_DRVRID</td>
<td>This error indicates that the <code>driverID</code> specified in the <code>tdiMemAllocSize()</code> function is not valid.</td>
</tr>
</tbody>
</table>

**Comments**

This routine is exported from the Tserver for use by the Drivers, and it can be called internally by the Tserver. It provides a method for determining how memory is used for message buffers in the Tserver-Driver Interface. Using this function, the Tserver and Driver can determine if the memory parameters specified at Driver registration time are sized appropriately. This routine also provides a mechanism for debugging message buffer configuration and handling problems.
**tdiQueueSize ( )**

This function will return the current status of the message buffers used for the TSDI. This routine returns a count of the messages in each of the four possible states: queued to the Tserver, queued to the Driver, owned by the Driver, or owned by the Tserver.

**Syntax**

```c
#include <tdi.h>

TDIReturn_t tdiQueueSize (
    TDIHandle_t driverID, /* INPUT */
    TDIQueue_info_t *queue_descriptor); /* OUTPUT */
```

**Parameters**

*driverID*

This is the unique identification number given to the driver when it registered with the Tserver via the `tdiDriverRegister()` routine.

*queue_descriptor*

This parameter returns the following information in the `TDIQueue_info_t` structure:

```c
typedef struct {
    int queued_to_driver;
    int queued_to_tserver;
    int allocd_by_driver;
    int allocd_by_tserver;
} TDIQueue_info_t;
```

Where the sum of all of these fields equals the total number of messages currently allocated for this TSDI. The fields are defined as follows:

*queued_to_driver*

This count specifies the number of message buffers that are currently queued to the Driver.
**queued_to_tserver**

This count specifies the number of message buffers that are currently queued to the Tserver.

**allocd_by_driver**

This count specifies the number of message buffers that are currently allocated to the Driver. Message buffers are allocated to the Driver if the Driver has performed a `tdiAllocBuffer()` or a `tdiReceiveFromTserver()` function call.

**allocd_by_tserver**

This parameter specifies the number of message buffers that are currently allocated to the Tserver. Message buffers are allocated to the Tserver if it has performed a `tdiAllocBuffer()` or a `tdiReceiveFromDriver()` function call.

**Return Values**

This function returns `TDI_SUCCESS` on success, and sets the return parameters to 0 on failure and returns one of the following negative values:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI_ERR_BAD_DRVRID</td>
<td>This error indicates that the <code>driverID</code> specified in the <code>tdiQueueSize()</code></td>
</tr>
<tr>
<td></td>
<td>function is not valid.</td>
</tr>
</tbody>
</table>

**Comments**

This routine is exported from the Tserver for use by the Drivers, and it can be called internally by the Tserver. It provides a method for determining how many message buffers are queued in either direction across the TSDI. Using this function the Tserver and Driver can provide a method of flow control by limiting the allocation of buffers and the sending of data based on the number of messages in the queues. This routine also provides a mechanism for debugging message buffer configuration and handling problems.
**tdiReceiveFromTserver()**

This function is called by the Driver to receive a populated message buffer from the Tserver. The buffer will be owned by the Driver until it is repopulated with a response message and sent back to the Tserver via the tdiSendToTserver() routine or until it is deallocated by the Driver via the tdiFreeBuffer() routine. This routine will only be able to receive a buffer if the routine tdiSendToDriver() has been previously executed by the Tserver to send a message to the Driver.

**Syntax**

```c
#include <tdi.h>

TDIReturn_t tdiReceiveFromTserver(
  TDIHandle_t driverID, /* INPUT */
  char **bufptr) /* OUTPUT */
```

**Parameters**

**driverID**

This is the unique identification number given to the Driver when it registered with the Tserver via tdiDriverRegister(). It allows the routine to identify correctly which buffer queue to use to access the message from the Tserver.

**bufptr**

This parameter is set to point to the start of the buffer returned by the tdiReceiveFromTserver() function call. If the tdiReceiveFromTserver() call is not successful, the function will set bufptr to NULL.

**Return Values**

This function returns **TDI_SUCCESS** on success, and bufptr is set to NULL and returns one of the following negative values on failure:
The `tdiReceiveFromTserver()` function receives a message that was sent by the Tserver across the TSDI. The buffer will be owned by the Driver until it is sent back to the Tserver via the `tdiSendToTserver()` routine or until it is deallocated via the `tdiFreeBuffer()` routine. This routine will only be able to receive a buffer if the routine `tdiSendToDriver()` has been previously executed by the Tserver to send a message to the Driver.

**Driver Notes**

The Driver is responsible for issuing a `tdiDriverRegister()` function call to specify the maximum number of bytes that can be allocated for message buffers by the Tserver or the Driver for this interface. The `driverID` returned by the `tdiDriverRegister()` routine must be used for the `tdiReceiveFromTserver()` routine to receive messages from the Tserver over the TSDI. The Driver is responsible for calling `tdiReceiveFromTserver()` to retrieve messages from its queues in a timely manner. The Driver must give the memory buffer back to the TSDI via the `tdiFreeBuffer()` routine after the message has been processed, or the message can be sent back to the Tserver via the `tdiSendToTserver()` routine.

**Warning**

The message buffer returned by this function should not be directly returned to the OS by the Driver. The Driver should return this buffer back to the TSDI as outlined above.

---

**TDI_ERR_BAD_DRVRIID**

This error indicates that the `driverID` specified in the `tdiReceiveFromTserver()` function is not valid. This may indicate that the Driver unregistered while the function was blocked waiting for a message from the Tserver.

**TDI_ERR_ESYS**

This error indicates that some form of system error has occurred. When this occurs, the TSDI will place an entry in the Error Log.
tdiSendToTserver()

This function allows the Driver to send a message buffer to the Tserver. The message will be queued until a corresponding tdiReceiveFromDriver() routine is called by the Tserver. A priority parameter is provided to put a message at the front of the queue. This routine is called by the Driver after a buffer has been allocated by the tdiAllocBuffer() routine and populated by the Driver.

Syntax

```c
#include <tdi.h>

TDIReturn_t tdiSendToTserver(
    TDIHandle_t driverID, /* INPUT */
    char *bufptr, /* INPUT */
    TDIPriority_t priority); /* INPUT */
```

Parameters

- **driverID**

  This is the value of the handle returned by the tdiDriverRegister() function call. This handle uniquely identifies the TSDI.

- **bufptr**

  This parameter is a pointer to the start of the buffer returned by the tdiAllocBuffer() function call or a tdiReceiveFromTserver() function call. After the tdiSendToTserver() routine completes, the Driver should no longer access the buffer.

- **priority**

  The priority is used to determine the priority class for the message. The default value, TDI_NORMAL_MESSAGE, should be used for all nonpriority messages, and TDI_PRIORITY_MESSAGE indicates that this is a priority message. Messages will be processed in FIFO order within their priority class, and priority messages will always be received by the Tserver before normal messages.

Return Values
This function returns **TDI_SUCCESS** on success, and returns one of the following negative values on failure:

**TDI_ERR_BAD_DRVRIID**
This error indicates that the **driverID** specified in the **tdiSendToTserver()** function is not valid.

**TDI_ERR_BAD_BUF**
This error indicates that the memory block pointed to by **bufptr** is not currently allocated from TSDI by the Driver.

**TDI_ERR_EINVAL**
This error indicates that the **priority** parameter contains an invalid value.

**TDI_ERR_NOT_YOUR_BUF**
This error indicates that the **driverID** specified did not match the **driverID** stored with this TSDI Buffer when the buffer was created via the **tdiAllocBuffer()** call. A Driver is only allowed to send TSDI Buffers that were originally allocated for this Driver.

**NOTE:** All TSDI Buffers allocated by a Tserver to be sent across the TSDI to a Driver are allocated with that Driver's ID. When this occurs, the TSDI will place an entry in the Error Log.

**TDI_ERR_ESYS**
This error indicates that some form of system error has occurred. When this occurs, the TSDI will place an entry in the Error Log.

**Comments**
This function sends a message from the Driver to the Tserver. The message will be queued until a corresponding \texttt{tdiReceiveFromDriver()} routine is called by the Tserver. Messages are queued in a First-In-First-Out manner, but a priority parameter is provided to override this mechanism and place this message at the front of the queue. This routine must specify a \texttt{bufptr} that has been allocated by the Driver via the \texttt{tdiAllocBuffer()} routine.

\section*{Driver Notes}

The Driver is responsible for issuing a \texttt{tdiDriverRegister()} function call to specify the maximum number of bytes that can be allocated for message buffers by the Tserver or the Driver for this interface. The \texttt{driverID} returned by the \texttt{tdiDriverRegister()} routine must be used to send a message to the Tserver via the \texttt{tdiSendToTserver()} function call. The Driver must allocate a message buffer via the \texttt{tdiAllocBuffer()} function call prior to calling \texttt{tdiSendToTserver()}, and the Driver is no longer responsible for that message buffer after \texttt{tdiSendToTserver()} has completed successfully.

\section*{Tserver Notes}

The Tserver is responsible for calling \texttt{tdiReceiveFromDriver()} to retrieve messages from its queues in a timely manner. The Tserver must give the memory buffer back to the TSDI via the \texttt{tdiFreeBuffer()} routine after the message has been processed, or send this message buffer back to the Driver via the \texttt{tdiSendToDriver()} routine.

\subsection*{tdiSetMessageFlowControl()}

This function will set the specific CSTA and/or ACS messages that the Driver wants the Tserver to flow control.

\textbf{Syntax}

\begin{verbatim}
#include <tdi.h>

TDIReturn_t tdiSetMessageFlowControl(
    TDIHandle_t   driverID, /* INPUT */
    short int    msgLevel, /* INPUT */
    SetFlowControl_t *msgs)  /* INPUT */

\end{verbatim}
Parameters

**driverID**

This is the unique identification number given to the driver when it registered with the Tserver via the `tdiDriverRegister()` routine.

**msgLevel**

This is the number of TSDI buffers **queued to the Driver** and/or **allocated by the Driver** at which the Tserver will initiate this flow control.

**msgs**

This is a pointer to a `SetFlowControl_t` structure. This structure contains fields for each possible message a client can send to a driver. If the driver wishes for the Tserver to flow control a particular message, then the corresponding field in this structure should be set to known zero. The `SetFlowControl_t` structure has the following format:

```c
typedef struct SetFlowControl_t {
    char alternateCall;
    char answerCall;
    char callCompletion;
    char clearCall;
    char clearConnection;
    char conferenceCall;
    char consultationCall;
    char deflectCall;
    char pickupCall;
    char groupPickupCall;
    char holdCall;
    char makeCall;
    char makePredictiveCall;
    char queryMwi;
    char queryDnd;
    char queryFwd;
    char queryAgentState;
    char queryLastNumber;
    char queryDeviceInfo;
    char reconnectCall;
    char retrieveCall;
    char setMwi;
    char setDnd;
    char setFwd;
};
```
char setAgentState;
char transferCall;
char eventReport;
char routeRegister;
char routeRegisterCancel;
char reRoute;
char routeSelect;
char monitorDevice;
char monitorCall;
char monitorCallsViaDevice;
char changeMonitorFilter;
char monitorStop;
char monitorEnded;
char snapshotDeviceReq;
char snapshotCallReq;
char escapeService;
char escapeServiceConf;
char sysStatReq;
char sysStatStart;
char sysStatStop;
char sysStatFilter;
char changeSysStatFilter;
char sysStatReqConf;
char acsOpenStream;

} SetFlowControl_t;

Return Values

This function returns **TDI_SUCCESS** on success, and sets the return parameters to 0 on failure and returns one of the following negative values:

**TDI_ERR_ESYS**
This error indicates a system problem.

**TDI_ERR_NO_MEM**
This error indicates the TSDI memory for this driver has been exhausted.
tdiGetMessageFlowControl ( )

This function will get the specific CSTA and/or ACS messages that the Driver wants the T server to flow control.

Syntax

```c
#include <tdi.h>

TDIReturn_t tdiGetMessageFlowControl(
    TDIHandle_t    driverID, /* INPUT */
    shortint *msgLevel, /* OUTPUT */
    SetFlowControl_t*msgs) /* OUTPUT */
```

Parameters

**driverID**

This is the unique identification number given to the driver when it registered with the T server via the tdiDriverRegister() routine.

**msgLevel**

This is a pointer that will receive the number of TSDI buffers queued to the Driver and/or allocated by the Driver at which the T server will initiate this flow control.

**msgs**

This is a pointer to a SetFlowControl_t structure. This structure is set to the value provided by the driver in the tdiSetMessageFlowControl() routine and contains fields for each possible message a client can send to a driver. If the driver wished for the Tserver to flow control a particular message, then the corresponding field in this structure should have been set to not zero. The SetFlowControl_t structure has the following format:

```c
typedef struct SetFlowControl_t {
    char  alternateCall;
    char  answerCall;
} SetFlowControl_t;
```

**TDI_ERR_BAD_DRVRID**

This error indicates that the `driverID` specified in the tdiSetMessageFlowControl() function is not valid.
char callCompletion;
char clearCall;
char clearConnection;
char conferenceCall;
char consultationCall;
char deflectCall;
char pickupCall;
char groupPickupCall;
char holdCall;
char makeCall;
char makePredictiveCall;
char queryMwi;
char queryDnd;
char queryFwd;
char queryAgentState;
char queryLastNumber;
char queryDeviceInfo;
char reconnectCall;
char retrieveCall;
char setMwi;
char setDnd;
char setFwd;
char setAgentState;
char transferCall;
char eventReport;
char routeRegister;
char routeRegisterCancel;
char reRoute;
char routeSelect;
char monitorDevice;
char monitorCall;
char monitorCallsViaDevice;
char changeMonitorFilter;
char monitorStop;
char monitorEnded;
char snapshotDeviceReq;
char snapshotCallReq;
char escapeService;
char escapeServiceConf;
char sysStatReq;
char sysStatStart;
char sysStatStop;
char sysStatFilter;
char changeSysStatFilter;
char sysStatReqConf;
char acsOpenStream;
Return Values

This function returns TDI_SUCCESS on success, and sets the return parameters to 0 on failure and returns one of the following negative values:

- **TDI_ERR_ESYS**: This error indicates a system problem.
- **TDI_ERR_NO_MEM**: This error indicates the TSDI memory for this driver has been exhausted.
- **TDI_ERR_BAD_DRVRIID**: This error indicates that the `driverID` specified in the `tdiGetMessageFlowControl()` function is not valid.

---

### tdiTrafficRegister()

This function allows a Driver (which provides CSTA services) to register itself with the Traffic Measurements Utility of the Tserver and requires that the driver must be registered with the Tserver (through `tdiDriverRegister()`). The Traffic Measurements feature is intended for CSTA services only and will return an error if the Driver has not registered with the Tserver for CSTA services (or has not registered with the Cserver.)

The return code from `tdiTrafficRegister()` will indicate to the caller whether the Traffic Measurements feature is enabled or disabled so the Driver will know whether or not to report traffic information. If the feature is turned off, the driver will still remain registered with the Traffic Measurements Utility. However, it should not report traffic data until it is told that the feature has been enabled. It will receive a TDI_ERR_TRAFFIC_OFF return code in this case. If the feature is enabled and all parameters are valid, then the return code will be set to TDI_SUCCESS.

The driver must specify an address of a callback function for the Traffic Measurements Utility to call when an administrator changes the enable/disable settings for the Traffic feature. There are two arguments to the callback function; `driverID` will be set to the value that was passed to `tdiTrafficRegister()` and...
cbParam will inform the driver whether measurements has been enabled or disabled.

Syntax

```c
#include <tdi.h>

typedef void(*TrafficMeasCB)
    (TDIHandle_t driverID, int cbParam);

TDIReturn_t tdiTrafficRegister(
    TDIHandle_t driverID, /* INPUT */
    TDIOfWeekType_t trafficType, /* INPUT */
    TrafficMeasCB callback ); /* INPUT */
```

Parameters

**driverID**
This is the unique identification number given to the Driver when it registered with the T server (or when it registered with the C server).

**trafficType**
This informs the Traffic Measurements Utility of the type of data that is being tracked. This argument must be set to TDI_PBX_DRIVER.

**callBack**
This is a pointer to a callback function that will be called by the T server when the Traffic Measurements feature or the Driver interface is enabled or disabled while a driver is registered with the Traffic Measurements Utility. The Traffic Measurement Utility will fill in the arguments to the callback function, *driverID* and *cbParam*. The *driverID* argument will contain the driver identifier for which this callback function is intended. This permits a Driver that registers multiple CSTA services with the T server to implement one callback function to handle all of its drivers. The *cbParam* argument will be set to one of the following values:
Return Values

This function returns **TDI_SUCCESS** when all parameters are valid and the Traffic Measurements feature is enabled. The return of **TDI_ERR_TRAFFIC_OFF** indicates that the registration was successful but the Traffic Measurements feature is disabled either for the system or for this driver interface and the driver should not send any traffic reports. On other failures, this function returns one of the following negative values:

- **TDI_TRAFFIC_ENABLED**
  
  This value indicates that Traffic Measurements has been enabled for this Driver interface and the Driver should start reporting traffic information.

- **TDI_TRAFFIC_DISABLED**
  
  This value indicates that Traffic Measurements has been disabled for this Driver interface and the Driver should stop reporting traffic information.

- **TDI_ERR_BAD_DRV RID**
  
  This error indicates that the *driverID* specified in the `tdiTrafficRegister()` function is not valid.

- **TDI_ERR_ESYS**
  
  This return code indicates that the Tserver could not allocate the resources necessary to register this Driver.
This function allows a Driver to report traffic information from its interface to the PBX (not its interface to the T server) to the Traffic Measurements Utility. It must use the `driverID` that was returned by the `tdiDriverRegister()` routine and must have registered with this utility by calling `tdiTrafficRegister()`.

### Syntax

```c
#include <tdi.h>

TDIReturn_t tdiTrafficReport(
    TDIHandle_t driverID, /* INPUT */
    TDITypeInfo_t trafficType, /* INPUT */
    TDIStatType_t statType, /* INPUT */
    TDIStatValue_t statValue ); /* INPUT */
```

### Parameters

**driverID**
This is the unique identification number given to the Driver when it registered with the Tserver.

**trafficType**
This informs the Traffic Measurements Utility of the type of data that is being tracked. This argument must be set to TDI_PBX_DRIVER.

---

**TDI_ERR_EINVAL**
This error indicates that the driver is not registered with the Tserver for CSTA Services or that a NULL callback function address was provided or that the trafficType argument does not contain the value, TDI_PBX_DRIVER.
**statType**

This parameter indicates which measurement the driver is reporting. This parameter is mandatory and must be set to one of the following:

- **TDI_RECEIVED_MESSAGE**
  This indicates that the driver received a message from the PBX and was able to service the request.

- **TDI_SENT_MESSAGE**
  This indicates that the driver successfully sent a message to the PBX.

- **TDI_REJECTED_MESSAGE**
  This should only be reported if the driver was unable to service this message due to a resource problem. When counting a message here it should not be included in the received or sent message counts.

**statValue**

This is a count of the number of messages for the given statType. It is expected that this will always be one but the driver may choose to process several messages before it reports the statistic, in which case, it would set this value accordingly. Note that waiting to report traffic data in volume instead of on a singular basis will skew the peak measurement and is not recommended.

**Return Values**

This function returns **TDI_SUCCESS** on success, and returns one of the following negative values on failure:
**tdiTrafficUnregister()**

This function allows a Driver to unregister itself with the Traffic Measurements Utility. It must use the `driverID` that was returned by the `tdiDriverRegister()` routine. This function must be called before the Driver unregisters with the Tserver.

### Syntax

```c
#include <tdi.h>

TDI_Return_t tdiTrafficUnregister (  
  TDIHandle_t     driverID, /* INPUT */  
  TDITrafficType_t trafficType); /* INPUT */
```

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDI_ERR_BAD_DRVRIID</td>
<td>This error indicates that the <code>driverID</code> specified in the <code>tdiTrafficReport()</code> function is not valid. This means that the Driver may no longer be registered with the Tserver.</td>
</tr>
<tr>
<td>TDI_ERR_NOT_REGISTERED</td>
<td>This error indicates that while the <code>driverID</code> is valid (that is, the Driver has registered with the Tserver using <code>tdiDriverRegister()</code>), it has not registered with the Traffic Measurements Utility using <code>tdiTrafficRegister()</code></td>
</tr>
</tbody>
</table>
Parameters

**driverID**
This is the unique identification number given to the Driver when it registered with the T server.

**trafficType**
This informs the Traffic Measurements Utility of the type of data that is being tracked. This argument must be set to TDI_PBX_DRIVER

Return Values

This function returns **TDI_SUCCESS** on success, and returns one of the following negative values on failure:

- **TDI_ERR_BAD_DRVRID** This error indicates that the **driverID** specified in the tdiTrafficUnregister() function is not valid. This means that the Driver may no longer be registered with the T server.

- **TDI_ERR_NOT_REGISTERED** This error indicates that even though the **driverID** is valid (that is, the Driver has registered with the Tserver using tdiDriverRegister() ) it has not registered with the Traffic Measurements Utility using tdiTrafficRegister().
**TDI_ERR_EINVAL**  
This error indicates that the *trafficType* argument does not contain the value, TDI_PBX_DRIVER.
TSDI and CSTA Message Interface
Header Files

All header files needed for TSDI development can be obtained from the TELEPHONY SERVICES SDK.

- tdi.h
- acs.h
- acsdefs.h
- csta.h
- cstadefs.h
Introduction

This chapter describes the interface provided by the Telephony Server for support of Driver-defined Operation, Administration, and Maintenance (OA&M) Services. The Telephony Server provides a simple OA&M interface between a client application and a Driver that has registered with the Tserver. This interface, from the Telephony Server's viewpoint, is just a block of data sent by an application to a driver and an event message sent in response by the driver back to the application. The format of the data is entirely up to the PBX Driver and the application. This will allow a customized OA&M client application to be developed by any vendor for their own Driver.
**tsrvDriverRequest()**

This function sends a `TSRVDriverOAMReq` message to the Driver specified by the `acsHandle`. A `TSRVDriverOAMConfEvent` will be returned from the Driver in response to this request. The application must receive the confirmation event via the `acsGetEventBlock()` or `acsGetEventPoll()` function.

**Syntax**

```c
#include <acs.h>
#include <tdrvr.h>

TSAPI
tsrvDriverRequest (ACSHandle_t acsHandle, /* INPUT */
  InvokeID_t invokeID, /* INPUT */
  unsigned char FAR *data, /* INPUT */
  _Int length); /* INPUT */
```

**Parameters**

* **acsHandle**
  This is the value of the unique handle to the opened ACS Stream returned by the function call. This handle is determined by the API Client Library and is unique to the ACS Stream being opened. Once the open function is successful, this handle must be used in all other function calls to the API. If the open is successful, the application is guaranteed to have a valid handle available upon return from this call. If the open is not successful, then the function return code will contain the cause of the failure.

* **invokeID**
  A handle provided by the application to be used for matching a specific instance of a function service request with its associated confirmation event. This parameter is only used when the Invoke ID mechanism is set for Application-generated IDs in the `acsOpenStream()`. The parameter is ignored by the ACS Library when the Stream is set for Library-generated invoke IDs.

* **data**
  A pointer to a data buffer the application is sending to the Driver.

* **length**
  The length of the data buffer pointed to by `data`. 
Return Values

This function returns the following values depending on whether the application is using library, or application-generated invoke identifiers:

*Library-generated Identifiers* - if the function call completes successfully, it will return a positive value, that is, the invoke identifier. If the call fails, a negative error (<0) condition will be returned. For library-generated identifiers, the return will never be zero (0).

*Application-generated Identifiers* - if the function call completes successfully, it will return a zero (0) value. If the call fails, a negative error (<0) condition will be returned. For application-generated identifiers, the return will never be positive (>0).

The application should always check the `TSRVDriverOAMConfEvent` message to ensure that the service request has been acknowledged and processed by the Telephony Server and the switch.

The following are possible negative error conditions for this function:

**ACSERR_APIVERDENIED**

This return indicates that the API Version requested is invalid and not supported by the existing API Client Library.

**ACSERR_BADPARAMETER**

One or more of the parameters is invalid.

**ACSERR_DUPSTREAM**

This return indicates that an ACS Stream is already established with the requested Tserver.

**ACSERR_NODRIVER**

This error return value indicates that no API Client Library Driver was found or installed on the system.

**ACSERR_NOSERVER**

This indicates that the requested Tserver is not present in the network.

**ACSERR_NORESOURCE**

This return value indicates that there are insufficient resources to open an ACS Stream.
TSRVDriverOAMConfEvent

This event is generated in response to the tsrvDriverRequest() function and provides the application with the confirmation event from the Driver.

Syntax

The following structure describes the format of the confirmation event received.

```c
typedef struct {
    ACSHandle_tacsHandle;
    EventClass_teventClass;
    EventType_teventType;
} ACSEventHeader_t;

typedef struct {
    ACSEventHeader_teventHeader;
    union {
        TSRVDriverConfirmationEventdriverConfirmation;
    } event;
    char heap[TSRV_DRIVER_HEAP];
} TSRVDriverEvent_t;

typedef struct {
    InvokeID_tinvokeID;
    union {
        TSRVDriverOAMConfEvent_tdriverConf;
    } u;
} TSRVDriverConfirmationEvent;

typedef struct TSRVDriverOAMConfEvent_t {
    int length;
    unsigned char FAR *data;
} TSRVDriverOAMConfEvent_t;
```
Parameters

\textbf{acsHandle}
This is the handle for the newly opened ACS Stream.

\textbf{eventClass}
This is a tag with the value TDRVRCONFIRMATION, which identifies this message as an Tserver Driver OA&M confirmation event.

\textbf{eventType}
This is a tag with the value TSRV_DRVEROAM_CONF, which identifies this message as a TSRVDriverOAMConfEvent.

\textbf{invokeID}
This parameter specifies the requested instance of the function or event. It is used to match a specific function request with its confirmation events.

\textbf{data}
A pointer to a data buffer the Driver is sending to the application.

\textbf{length}
The length of the data buffer pointed to by data.

\textbf{TSRVDriverOAMEvent}
This event can occur at any time (unsolicited) and is sent by the Driver as defined by the Driver OA&M scheme.

\textbf{Syntax}
The following structure describes this event.

\begin{verbatim}
typedef struct
{
   ACSHandle_t acsHandle;
   EventClass_t eventClass;
   EventType_t eventType;
} ACSEventHeader_t;

typedef struct
{
   ACSEventHeader_t eventHeader;
   union
   {
      TSRVDriverUnsolicitedEvent driverUnsolicited;
   } event;
   char heap[TSRV_DRIVER_HEAP];
}
\end{verbatim}

TSDI.PDF — Avaya CT 1.2

Issue 1 — December 2002
} TSRVDriverEvent_t;

typedef struct TSRVDriverOAMEvent_t
{
    int       length;
    unsigned char  FAR   *data;
} TSRVDriverOAMEvent_t;

Parameters

acsHandle
This is the handle for the newly opened ACS Stream.

eventClass
This is a tag with the value TDRVRUNSOLICITED, which identifies this message as an ACS unsolicited event.

eventType
This is a tag with the value TSRV_DRIVEROAM, which identifies this message as an TSRVDriverOAMEvent.

data
A pointer to a data buffer the Driver is sending to the application.

length
The length of the data buffer pointed to by data.
OA&M Header Files

This section describes the C Language header file that defines the interface provided by the Telephony Server for support of Driver-defined Operation, Administration, and Maintenance (OA&M) Services. All header files needed for TSDI development can be obtained from the TELEPHONY SERVICES SDK.

- drvrdefs.h
- tdrvr.h
Implementation Notes

This section describes what a Driver must implement specifically for the Windows NT platform. This section provides coding examples of these requirements and also some hints on porting a Driver from NetWare to Windows NT.

Overview

On the Windows NT platform the T server is an executable that runs as an NT Service on both NT Workstations and Servers, Version 3.51 and higher. Drivers are implemented as DLLs which the T server loads into its address space for sharing of memory across the TSDI. This requires the Driver to export two functions for starting and stopping the driver. These functions are described in more detail in this section.

The T server service can be administered to start up automatically when the NT system is rebooted or manually. Likewise, since the T server is responsible for starting and stopping drivers it also provides a similar capability to automatically load drivers or to require a manual start for a driver.

The T server provides this interface through its OA&M application which gives administrators the options of adding a driver name to a list, enabling or disabling the autoload setting for each driver, manually loading or unloading each driver or to remove this driver from the list. If the autoload setting is enabled the T server will load this driver each time the T server service is started.

Since the T server and Driver are loaded into the same address space and since the T server controls the loading and unloading of the Driver, it may be difficult for the Driver to exit gracefully if it encounters a serious problem. In NetWare a Driver could call exit() and terminate without affecting the T server. In NT, however, calling exit() will cause the T server to terminate as well. Because of this, the T server provides an additional TSDI API which the Driver should call when it wants to terminate itself gracefully. This API, tdiDriverUnload(), tells the T server to perform the unload procedure on the specified driver. The manual page for this new API is included below. This new function is only available on the Windows NT platform since the loading and unloading mechanism is different from the NetWare platform.
The Telephony Services SDK also provides a DLL (insttsdr.dll) which the Driver can use during its installation to put its name into this list. Refer to the SDK for details on how to use this install feature. The Driver is not required to use this dll; it is only provided as a convenience so that the administrator will not have to run the Tserver's OA&M application after installing the Driver.

**Driver Requirements**

Telephony Services for NT requires Drivers to export two functions that the Tserver will use to load and unload the Driver; `tdiStartDriver()` and `tdiStopDriver()`. This allows the Tserver to load or map the Driver into its address space and then start the Driver. For those porting a Driver from NetWare the `tdiStartDriver()` function is the Driver's main() function which was run as a result of the load nlm console command. The `tdiStopDriver()` function is the Driver's unload function which was run as a result of the unload nlm console command.

The reason for these new functions is to allow the Tserver to map the driver into its address space and start up the driver while still being able to process other requests. The Tserver will take the driver name (including the extension, e.g., sim.dll) and call `LoadLibraryEx()` with this name. The operating system will map this driver into the same address space of the Tserver executable and will cause the Driver to receive a `DLL_PROCESS_ATTACH` message. If the Driver were to perform its complete initialization at this point the Tserver would be blocked waiting for the Driver to finish. It would also not know if the Driver initialization failed. The handling of a Driver failing to start up is discussed below. It is requested that the Driver do very little when it receives the `DLL_PROCESS_ATTACH` message; perhaps just initialize some variables. If the `LoadLibraryEx()` succeeds the Tserver will spawn a thread to call the Driver's `tdiStartDriver()` function.

The Tserver will call the `tdiStopDriver()` function when the administrator requests an unload of a driver or if the Tserver service is being stopped. This function tells the Tserver when the Driver is completely gone so that the Tserver can then free or unmap this library.

“Coding Examples” on page 7-8 contains coding examples for supporting these new functions.
Implementation Notes

tdiStartDriver()

This function allows the Tserver to start the Driver automatically when the Tserver service is started or when the administrator manually loads a Driver. It should not return until the Driver has completely initialized and is ready to process requests or it should return if the Driver has encountered an error that prevents it from initializing successfully.

Syntax

Boolean tdiStartDriver( void );

Return Values

This function returns TRUE when the Driver has successfully completed its initialization. A return code of FALSE indicates that the Driver could not load properly. If the Driver fails to load properly the Tserver will call the Driver's tdiStopDriver() function. It must do this to ensure that all of the Driver's threads are gone before it can free or unmap the driver from the address space. The Driver can cause the Tserver to abnormally terminate if it still has something running after the Tserver frees it (using FreeLibrary()).

tdiStopDriver()

This function allows the Tserver to stop the Driver when the Tserver service is stopping or when the administrator manually unloads a Driver. It should not return until the Driver has completely unloaded or it should return if the Driver has encountered an error that prevents it from stopping successfully.

Syntax

Boolean tdiStopDriver( void );

Return Values

This function returns TRUE when the Driver has successfully unloaded, meaning that all of its threads are gone and all of its resources have been deallocated. At this point the Tserver can unmap the Driver from the address space by calling FreeLibrary(). It is very important that the Driver does not return TRUE if any of its threads are still running. If the Driver has something running after the Tserver frees it, the Tserver service will abnormally terminate.

A return code of FALSE indicates that the Driver could not unload properly. If the Driver fails to unload properly the Tserver will not free the library in case there is still something running in the Driver. The Tserver will place an error message in its Error Log indicating that the Driver failed to unload properly.
tdiDriverUnload

This function allows a Driver to request that the Tserver perform the unload procedure for the Driver. This permits the Driver to exit gracefully in situations where it has encountered a serious error and cannot function properly. Upon receipt of this request, the Tserver will call the Driver's tdiStopDriver() function and expects it to behave as described in the tdiStopDriver() manual page.

Syntax

```c
#include <tdi.h>
TdiReturn_t tdiDriverUnload( const char*driver_name );
```

Parameters

**driver_name**

This string contains the name of the PBX Driver DLL. It is the same name that is used to load and unload the Driver and must include the extension; for example, "driver.dll". If the Driver installation uses the insttsdr.dll provided by the Telephony Services SDK to put its name into the Tserver's administered list of drivers, use the same name in the call to tdiDriverUnload(). Refer to the SDK for details on how to use this install feature.

Return Values

This function returns **TDI_SUCCESS** if the Tserver was able to start the unload process by calling the Driver's tdiStopDriver() function. It does not mean that the Driver has returned from its unload function. tdiDriverUnload() returns the following negative value on failure:

**TDI_ERR_ESYS**

This error indicates that some form of system error occurred while the Tserver was initializing. When this occurs, the Tserver will place an entry in the Error Log.

Comments

This function causes the Driver's tdiStopDriver() function to be called and then returns. The Tserver will wait for the tdiStopDriver() function to return a TRUE or FALSE. If it returns **TRUE** then the Driver has successfully unloaded, meaning all of its threads are gone and all of its resources have been deallocated. At this point the Tserver will unmap the Driver from the address space by calling FreeLibrary(). It is very important that the Driver does not return **TRUE** if any of its threads are still running. If the Driver has something running after the Tserver frees it, the Tserver service will abnormally terminate.

If tdiStopDriver() returns **FALSE**, this indicates that the Driver could not unload properly. If the Driver fails to unload properly the Tserver will not free the library...
in case there is still something running in the Driver. The Tserver will place an error message in its Error Log indicating that the Driver failed to unload properly.

### Thread Priorities

The Tserver runs at the default normal priority, not high priority or real time. It does not change its threads priorities (i.e., it does not call the Windows API, SetThreadPriority()). It is highly recommended that the Driver also run at the same priority level. If it were to run at a higher priority or real time the Tserver may not be able to process messages as it should.

Note: Windows NT tasking settings (control panel/system/tasking) may have more of an effect on the threads running in the processes than the thread priorities. The tasking settings change the responsiveness of the foreground and background applications.

### Compiling and Linking

The recommended compiler is Visual C++ 2.2 or greater.

The TSDI library routines (tdiDriverRegister(), tdiAllocBuffer(), etc.) should be linked by the Driver by including tdi.lib in the Project Files list. The header file, tdi.h, includes the definitions needed for these library routines.

### Porting from NetWare to NT

This section is included for those who are porting a Driver from the NetWare platform to Windows NT. The information is only included here as suggestions for implementation and is by no means meant as requirements.

### Platform Differences

The main consideration in porting a PBX driver from the NetWare platform to the Windows NT platform is protecting access to data structures. If your PBX driver depends on the fact that NetWare is a non-preemptible operating system, and access to data structures is protected by exclusive control of the CPU then the majority of the port will be ensuring data structures are protected through the use of critical sections and/or semaphores.

A secondary consideration in porting from NetWare to Windows NT is translating OS specific calls in NetWare to similar or equivalent calls in NT. Some main areas to consider are thread differences, memory allocation and semaphores.
Threads

The main differences with threads between NetWare and NT are the thread's function declaration, the different function call to spin off the thread and the requirement of a return value by NT. All of these are fairly trivial to port.

Memory allocation

Calls such as Alloc() and Free() in NetWare can easily be converted to heap allocation in Windows NT with macros that use HeapAlloc() and HeapFree() along with some initialization of the heap. The heap can also be initialized to automatically resize if memory needs increase.

Semaphores

Windows NT API calls for semaphores can also be easily called via macros that look like the NetWare calls in order to minimize porting considerations. Some of the calls to map would be:

- OpenLocalSemaphore() to CreateSemaphore()
- WaitForLocalSemaphore() to WaitForSingleObject() (with INFINITE wait)
- TimedWaitForLocalSemaphore() to WaitForSingleObject()
- SignalLocalSemaphore() to ReleaseSemaphore()
- ExamineLocalSemaphore() - NT does not have an equivalent to this and you would need to create macros that contain a data structure with a count that gets incremented and decremented with waits and signals to the semaphore.

Server Console Interfaces

Driver Server Console interfaces in NetWare are usually integrated as part of the driver NLM code and are displayed either as ASCII screens or NWSNUT screens on the Server Console. Windows NT offers the opportunity to provide a GUI interface, and since Telephony Services runs as an NT service, any Server Console interfaces must be independent applications that are run by a user when logging onto the NT server (Telephony Services runs independently of any users logged onto the NT server, thus so do any drivers that it loads).

Porting existing NetWare Server console screens

Perhaps the easiest way to port existing NetWare Server Console screens is to create a GUI application that loads the driver dll and uses a shared data section to communicate between the driver dll and the GUI application. This allows explicit access to data and the only porting issues are the differing mechanisms for displaying the data and controlling access to the data between the driver dll and the GUI application. Data access can be controlled through the use of critical sections or semaphores.

Issue 1 — December 2002
Tips on porting...

For the Driver DLL:

Creating a shared data section - in the driver dll create a shared data section to declare any variables that will be accessed by both the dll and the GUI application.

```c
#pragma data_seg(".shrdata."

// declare and INITIALIZE any shared data here

int SharedVariable = 0;

#pragma data_seg()
```

Create a DEF file and make the data section shared.

```c
SECTIONS
  .shrdata READ WRITE SHARED; Share .shrdata segment
```

Export the variables so they can be accessed by the GUI application.

```c
__declspec(dllexport) int SharedVariable;
```

Protect access to any shared data with a shared critical section or semaphore.

For the GUI application:

Import the variables so they can be accessed by the GUI application.

```c
__declspec(dllimport) int SharedVariable;
```

Protect access to any shared data with a shared critical section or semaphore.

Make sure the GUI application loads the driver DLL from the same directory that the Telephony Server has already loaded the DLL from.
Coding Examples

This section contains coding examples of the DllMain(), tdiStartDriver() and tdiStopDriver() functions that are required of a Driver running on NT.

DllMain()

/*!DllMain()
 *
 *This routine is called by NT when the dll is loaded 
 *or unloaded.
 * */
_declspec(dllexport)
BOOL WINAPI DllMain(HINSTANCE hinstDll, 
                   DWORD fdwReason, 
                   LPVOID lpvReserved)
{
    switch(fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        // Do minimal processing here
        // Threads spun off from here will
        // NOT execute until this function
        // call completes.
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        // The library is being freed. Any
        // cleanup should have been done by
        // the tdiStopDriver() function.
        default:
            break;
    }
    return(TRUE);
}

tdiStartDriver()

/*!tdiStartDriver()
 *
 *This routine is called by the Tserver after it loads the
driver library. This is the initialization routine of the Driver and should not return until the Driver is completely up and running or an error is encountered that will prohibit the Driver from initializing. If the Driver returns a failure (FALSE) from this function the Tserver will turn around and call the Driver's tdiStopDriver() function. It must do this to make sure the Driver is completely gone before it can free the driver library.

```c
__declspec(dllexport) BOOL tdiStartDriver()
{
    int completionCode;

    tdiLogError(DRIVER_NM, TRACE, ERR_NO, 0,
            "tdiStartDriver has been called. \n            About to startup the <DRIVER_NAME>"
    );

    /*
    * driverInitialize() does not return until the driver is completely up and running or if an error has occurred.
    */
    completionCode = driverInitialize();

    if(completionCode == SUCCESS)
    {
        /* The driver has completely initialized and is ready to handle client requests.
        */
        return(TRUE);
    }
    else
    {
        /* The driver has encountered an error and cannot load successfully. The driver will expect to receive a tdiStopDriver() request at this point.
        */
        return(FALSE);
    }
}
```

tdiStopDriver()

/*tdiStopDriver()
 */
This routine is called by the Tserver before it frees the driver library. This is the unload function of the Driver and should not return until the Driver is completely unloaded. If it encounters an error that prohibits it from unloading properly it should return an error (FALSE) from this function.

```c
__declspec(dllexport) BOOL tdiStopDriver()
{
    int completionCode;

    tdiLogError(DRIVER_NM, TRACE, ERR_NO, 0,
                "tdiStopDriver has been called.  "\n                About to unload the <DRIVER_NAME>");

    completionCode = driverUnload();

    if(completionCode == SUCCESS)
    {
        /* The driver has completely unloaded. It can no longer handle client requests. */
        return(TRUE);
    }
    else
    {
        /* The driver has encountered an error and cannot unload successfully. The driver library will not be freed. */
        return(FALSE);
    }
}
```
Issue/Release History

- Issue 1.1 of the TSDI is the result of design changes during the initial development of the Telephony Server. The issue 1.1 document discusses primarily the AT&T G3 PBX Driver use of the TSDI.

- Issue 1.2 has been updated to include more information for use by outside PBX Driver vendors. The TSDI function calls have remained the same. Discussions have been added on processing ACS and CSTA messages and how to handle private data, and some basic coding examples are also included.

- Issues 1.3 and 1.4 contain minor editing and clarification changes.

- Issue 1.5 was updated with manual pages on two new TSDI functions: `tdiGetSessionIDInfo()` and `tdiMapInvokeID()`. The routine `tdiGetSessionIDInfo()` allows a driver to obtain detailed information about the client for a given session ID. The Tserver maps all application invokeIDs to unique invoke IDs created by the Tserver. The driver only receives the invokeID created by the Tserver. The routine `tdiMapInvokeID()` allows a driver to determine the actual application invoke ID.

- Issue 1.6 was updated to include discussions on flow control that have been added to the official Telephony Server Version 1.1.a.0.5. Flow control mechanisms in this release of the Telephony Server include the ability of a driver to selectively request which CSTA or ACS messages get flow controlled and at what level of TSDI space being used this flow control takes effect. The Tserver also contains a higher level of flow control at which all incoming client messages are rejected when the amount of TSDI space being used by a driver exceeds the defined TSDI High-Water Mark. The "Telephony Server Flow Control Of TSDI Messages" has been updated with descriptions of Telephony Server flow control. Issue 1.6 also provides manual pages for the following new TSDI interface routines: `tdiSetMessageFlowControl()` and `tdiGetTDISize()`. Issue 1.6 also provides a manual page for the TSDI routine `tdiLogError()`.
- Issue 2.0 has been updated to reflect changes to Release 2 of Telephony Services. These include the addition of a `tdiGetTservVersion()` function call which allows a driver to determine the version of the Tserver that it is communicating. Issue 1.6 of the document had the following errors that were also corrected: the `tdiDriverUnregister` routine is non-blocking. This was changed in the 1.1a release of the Telephony Server. Also, version 1.6 stated that the security option, `TDI_NO_SECURITY`, would check a user's login and password. If a driver registers with `TDI_NO_SECURITY`, then a client may open a stream without a valid login or password.

- Issue 2.0 also contains an update to the driver registration routine, `tdiDriverRegister`. In release 2, this routine allows the driver to indicate which version of the TSAPI protocol it can speak so the mechanism of TSAPI protocol version negotiation can occur in release 2.

- Issue 2.1 reflects solely the production changes (cover and footer information) required to prepare this document for the November 1994 Alpha Delivery to Beta ISVs.

- Issue 2.2 reflects additional changes for Release 2 Version Control. This includes TSDI Version Control, TSAPI Version Control, and Private Data Version Negotiation.

- Issue 2.3 includes three new APIs for Traffic Measurements. When a PBX Driver registers with the Tserver for CSTA services then it must also register with the Tserver for Traffic Measurements reporting.

- Issue 2.4 includes traffic measurement coding examples. Issue 2.4 also provides updated manual pages for `tdiGetSessionIDInfo()`. This function returns slightly different structure depending on whether the driver registered for TSDI version 1 or version 2.

- Issue 2.5 corrects the information in Issue 2.4 on the `tdiGetSessionIDInfo()` function. Rather than returning a different structure from the same function, a new function called `tdiGetSessionInformation()` was added to TSDI version 2.

- Issue 2.6 contains editorial enhancements.

- Issue 2.7 includes a new section, Implementation Notes, specific to the operating system.

- Issue 2.8 contains editorial enhancements.

- Issue 2.9 includes a new manual page, `tdiDriverUnload`, for Windows NT only.

- Issue 2.10NT contains editorial changes and the addition of NT to the issue number to readily identify it as the NT version of the document.

- Issue 2.11NT contains editorial changes.

- Issue 2.12NT contains editorial changes and the addition of NT to the issue number to readily identify it as the NT version of the document.
Issue/Release History

- Issue 2.13NT contains two new technical descriptions: Telephony Server Flow Control of TSDI Messages, and tdietMessageFlowControl.

- Issue 2.14NT contains editorial changes.

- For CVCT Release 3.30, Version 1 through CVCT R9.1 V1, release notation elements were changed (editorial changes only).

  NOTE:
  As a result of a change to the CVCT release numbering scheme, Release 3.30 was succeeded by CVCT Release 9.1.

- For CVCT Release R9.5 V1 the document was rebranded as Avaya and release elements were changed (editorial). Release 9.5 V1 introduced one technical change for tdiRegisterDriver as follows:
  — If the Telephony Services Security database (SDB) is disabled by administration, and if the driver registers with driver_security TDI_CSTA_SECURITY, the CVCT Telephony Server will not do CSTA device-level permission validation because no permissions are available; hence the effective registration will be TDI_LOGIN_SECURITY.

- For CVCT Release R10.1 V1 release notation elements were changed (editorial changes only).

- For CVCT Release R11.1 V1 this document is reissued to accommodate technical changes driven by the seven digit DEFINITY Dial Plan expansion, which expands the maximum extension length on the dial plan from five to seven digits.

- For Avaya Computer Telephony 1.2 this document is reissued to reflect the new product name (Avaya Computer Telephony) and release numbering.

Issue 1 — December 2002