µ §

Table of Contents

1. Introduction

This document is the external specification of the Telephony Services Driver Interface for the NetWare® Telephony Services product. The Telephony Server provides service for Computer-Supported Telecommunications Applications (CSTA) in a Novell NetWare® environment. The Telephony Server consists of three parts, the Application Programming Interface (API), the Tserver, and the PBX Driver.

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 Novell network with the Tserver.

The Tserver is a NetWare Loadable Module (NLM) that is responsible for routing messages between the application workstations on the network and the PBX Driver. The PBX Driver is an NLM that 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 NLM and any vendor's Driver NLM.

This document specifically covers the function call interface to pass the messages between the NLMs, the structure of the messages, and requirements for vendors that must be followed when writing a PBX Driver that adheres to the interface.

Section {1} 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.

Section {3} describes the architecture of the Telephony Server and the communication between its major components, 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.

Section {4} gives a detailed description of the Telephony Services Driver Interface from the PBX Driver perspective.

Section {5} discusses compiling and linking the Driver.

Section {6} provides coding examples which can be used as the basic skeletal outline of a Driver which will be using the TSDI to process CSTA messages.

The remaining sections define the specifics of the Telephony Services Driver Interface.

1.1. Overview

The Telephony Server provides the desktop integration of telephones and personal computers by exporting a Computer-Supported Telecommunications Applications (CSTA) API over a Novell® NetWare® network. Figure 1 provides a high level view of a simple Telephony Server configuration. Each phone and associated computer represents a workstation on someone's desktop, and workstations can run applications that are integrated with the local phone. The application will use the CSTA API supported by a native library loaded on the workstation. The CSTA API supported by the Telephony Server is based on the European Computer Manufacturers Association (ECMA) CSTA standard. The library supporting the CSTA API exchanges messages that represent API function calls and parameters over the Novell network with the Tserver. The Tserver is a NetWare Loadable Module (NLM) that resides on the Telephony Server. The Tserver is responsible for routing messages between the applications on workstations connected to the network, and the PBX Driver. The PBX Driver is an NLM that 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 which supports CTI links via the TSDI must define the physical implementation for a CTI link (i.e., the CTI link could be one or more physical links). The Telephony Services Driver Interface is an open interface that specifies how messages are passed between the Tserver NLM and any vendor's Driver NLM. Figure 1 shows one link to one switch. Note that the Telephony Server (actually, the PBX driver) can have multiple links, possibly terminating on different switches.

µ §

Figure 1: Telephony Server Connectivity

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**.

1.2. Terminology

API — *Applications Programming Interface.* The API specifies the access methods 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.

CSTA — *Computer-Supported Telecommunications Applications.*

CTI — *Computer Telephony Integration.*

NLM — NetWare Loadable Module - This is a module that can be dynamically loaded on the NetWare Operating System.

OA&M *Operations, Administration and Maintenance.* A module that provides the maintenance and administration interface.

PBX Driver The vendor dependent software specific to switch control. The PBX Driver is the service provider portion of the Telephony Server.

PDU *Protocol Data Unit.* A data object exchanged between the Telephony Server and the client application.

Tserver The specific module that manages the routing of CSTA requests and responses between a client application and the appropriate PBX driver. The Tserver NLM is part of the Telephony Server.

Telephony Server The Telephony Server supports CSTA in a Novell NetWare environment. The Telephony Server may also refer to a Novell file server that has been loaded with and runs the Telephony Server software.

TSDI Telephony Services Driver Interface: A connection between a Driver NLM and the Tserver NLM using a function call interface to pass messages between modules.

1.3. Related Documents

This document assumes that the reader is familiar with the documents listed below.

TSAPI *API Definition and Specification for the R1.0 NetWare® Telephony Server*, T.A. Anschutz, and J.R. Garcia.

[ECMA/52] *Technical Report ECMA/52, Computer Supported Telecommunications Applications (CSTA),* European Computer Manufacturers Association.

[ECMA-179] *STANDARD ECMA-179, Services For Computer Supported Telecommunications Applications (CSTA),* European Computer Manufacturers Association.

[ECMA-180] *STANDARD ECMA-180, Protocol For Computer Supported Telecommunications Applications (CSTA),* European Computer Manufacturers Association.

2. Document Updates

2.1. Reason for Reissue

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, how to handle private data, and some basic coding examples are also included.

2.2. Change History

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

3. Description

3.1. Telephony Server Architecture

The Telephony Server provides a platform for the desktop integration of telephones and personal computers. The personal computers or workstations can run CSTA applications that are integrated with their user's phones. The CSTA applications will use a library (the CSTA API) as a client communicating with a Telephony Server running on a NetWare® file server that has advertised for CSTA services on the network. The Telephony Server will transport client requests over a switch vendor specific link to provide switch integration. Multiple applications may access Telephony Services on the same workstation simultaneously, and multiple workstations may request services from the same Telephony Server. The Telephony Server provides a separate set of generic OA&M services. The OA&M application will use a separate native library (the OA&M API) as a NetWare client for communication with the Telephony Server. The OA&M API provides a basic interface for passing, what to the Telephony Server appears to be a character array, between the OA&M application and the Driver. The format and interpretation of these messages is defined entirely by the OA&M and the Driver. The Telephony Server supports one or more OA&M interfaces per PBX Driver (this is defined by the Driver). Figure 2 shows a block diagram of the Telephony Server architecture.

Figure 2: Block Diagram of Telephony Server

The Telephony Server runs under NetWare (the large box in Figure 2). The two main Telephony Server modules that run under NetWare include:

Tserver Manages 'Telephony' and 'OA&M' requests from clients (over the LAN). The Tserver will confirm that each client is administered for the requested service, CSTA or OA&M messages and authenticate some CSTA (but **not** OA&M) requests, and route the requests to the appropriate PBX Driver.

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 their own OA&M 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 Telephony Services Driver Interface 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 NetWare 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, i.e., each ***tdiDriverRegister*** generated a different driver ID which the Driver then uses in ***tdiReceiveFromTserver*** calls and all CSTA client requests which come over ACS streams that where 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.

3.2. Telephony Server Communication 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.

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 that 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.*

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 section {4.4}.

3.3. 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.

3.3.1. 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, NetWare® 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 are completely separate interfaces, and message traffic across these interfaces are completely independent.

3.3.2. 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 Tserver NLM. The PBX Driver must always be prepared to negatively acknowledge client application requests that the Driver does not understand.
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 (i.e 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 section 4.

3.3.3. Driver Termination

The PBX Driver should always unregister with the Tserver before exiting (unloading), or the Driver can unregister with the Tserver any time it wants to stop handling CSTA or OA&M requests. The Tserver 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 can not 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.

3.4. Telephony Server Multiplexing

The telephony server **Provides No Multiplexing Services** for the Driver. An ACS stream is opened to a Driver (i.e. CTI-LINK name as generated by a Driver registration), not to a device. The application must know ahead of time which CTI-LINK to open a stream to. When CSTA reqeuests are received for a device, the request is received by the Tserver over an ACS stream that terminates at a CTI-LINK (driver). The request is forwarded to the driver if and only if the security data base indicates the device in the request can be controlled by the user (login)

which opened the ACS stream and that the CTI-LINK that the ACS stream terminates on can control the device. If the security data base indicates that more than one CTI-LINK can be used to control a device, **the Tserver does not select a CTI-LINK** to forward a CSTA request to. The Tserver always forwards reqeusts to the CTI-LINK (driver) 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 reqeust for the driver.

In the security data base, users (logins) are given permission to control devices and all the CTI-LINKs which can support (control) a device are indicated.

3.5. 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 NetWare® Login on the file server the Tserver is loaded on. The lowest level of security a driver can register with is **TDI_NO_SECURITY** which means the Tserver will only validate the login ID and password provided in an *acsOpenStream()* call are valid according to the Bindery on the file server. This may be useful when developing the driver. The next level of security a driver can request during the registration is **TDI_LOGIN_SECURITY** which requires the same NetWare® Login and password on the file server and an entry in the TServer Security Data Base. This may be useful for the driver OA&M support. The highest level of security, **TDI_CSTA_SECURITY**, encompasses the requirements of the previous two levels and in addition validates CSTA requests against the Security Data Base administration.

3.5.1. Client Access Security Levels

The Telephony Server supports a Security Database which allows the administration of users, groups of users , users worktops, devices, list of devices, and list 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 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 (i.e. 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 Tservers/PBXs which support this device.

3.5.2. Security Issues and Class of Service

The Telephony Server defines four classes of service which 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, device monitor** or **query** the device or set of devices associated directly with a users 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 users home worktop location.
There are two flavors of the home worktop class of service. The first is based on the assumption that the network and node field (LAN Address) of an SPX packet can be believed, the other assumes that these fields are not secure and can easily be faked in a packet sent to the Telephony Server.

If the administration is setup 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 home worktop privilege over a device:

When a user is logging in from their home workstation, they get privilege over the set of devices defined by their worktop record. When a user logs in from a workstation other than their home workstation, they get privilege over their home workstation set of devices **as well as** the set of devices associated with the workstation they 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, they get privilege over the set of devices defined by their 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. There are three types of monitoring privileges which can be granted to a user:

- 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 is can be given permission to call monitor calls. Because calls are not known in advance, a specific list of calls which might be monitored can not 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.

3.6. Driver OA&M

3.6.1. Using the Tserver as a Transport

The PBX Driver vendors can implement a client or NLM based application using the generic Telephony Server Driver OA&M API defined in section 10 for driver OA&M. The Tserver would act as the transport mechanism in this case. To use the Tserver transport mechansim 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 defined to be a 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 Tserver will provide administration options on User Records that will provide access to the OA&M link for specific users. i.e. each user in the Tserver 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 NLM, from a seperate Driver OA&M NLM 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.

3.7. Error Log Interface

A common error log will serve both the Tserver and the PBX driver. The interface to the error log is always through the Telephony Server OA&M client application. It will support a standard function call interface so errors will have a uniform appearance in the error log. The error log interface will provide six severity levels TRACE, CAUTION, AUDIT_TRAIL, WARNING, ERROR, FATAL (see section {4.5} below) for errors, and will include the date, time, location of the error, a specific error code and supporting text for each error. There are three possible destinations for each severity level: the NetWare® system console, the Tserver OA&M application, and the error log file. These destinations can be set for each of the six severity levels through the Tserver OA&M client. Whenever possible, the PBX driver should log errors in the common error log with the appropriate severity. The PBX Driver authors are free to ignore this error log interface and provide their own error logging mechanism, but their error messages will not be integrated with the Telephony Server.

4. Functional Description

4.1. PBX Driver to Tserver Interface

The TSDI provides a function call interface providing 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 section {4.4}). 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 section 7.3) from the TSDI; they can not be directly allocated from the NetWare® Operating System. The PBX Driver should always unregister with the Tserver before it is unloaded.

The following sections provide a brief description of the function call interface provided by the Telephony Services Driver Interface routines. See Section 7 for a complete specification of the interface routines.

4.1.1. Driver Registration

4.1.1.1. 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 section {4.4} below. A separate connection must be created for each type of service the PBX Driver will provide (e.g. one for CSTA services and one for OA&M services). Each CTI Link supported by the Driver for which the Driver wants services advertised for must be registered seperately with the Tserver, i.e 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 physicaly links. A Driver should register separately for each CTI Link that it will support.

The Driver must use the Telephony Services Driver Interface routine, tdiDriverRegister(), to establish a connection between the Driver and the Tserver. When a PBX Driver registers via the tdiDriverRegister() routine, the Tserver NLM creates a separate TSDI instance by allocating the resources from the NetWare® OS for this Telephony Services Driver Interface (TSDI), and begin Service Advertising on behalf of the PBX Driver. The Telephony Server treats each tdiDriverRegister(), or TSDI instance completly independently. From the Telephony Servers point of view each tdiDriverRegister() represents a different Driver, even though multiple registrations may have been done by the same Driver NLM.

The Tserver NLM will apply the ***vendor_name, service_name*** and ***service_type*** parameters provided by the Driver in the tdiDriverRegister() routine to the NetWare AdvertiseService() routine (see **[NLMREF-II]**). The AdvertiseService() routine will inform (CSTA or OA&M) clients that the Driver is available to perform services. The Tserver NLM will also store the ***driver_name*** parameter provided by the PBX Driver in the tdiDriverRegister() routine for maintenance purposes. Version information must be specified in the tdiDriverRegister() routine so that the PBX Driver can guard against version compatibility problems with the Tserver. Because only one version of the TSDI currently exists, the ***version*** field must always be set to **TSDI_VERSION.** The Telephony Services Driver Interface monitors TSDI buffer usage by both the PBX Driver and the Tserver NLMs. The PBX Driver can specify a maximum number of bytes that can be allocated (by the PBX Driver and the Tserver) for TSDI messages via the ***buffer_descriptor*** parameter of the tdiDriverRegister() routine. See section 4.1.1.2 below for a detailed description of how memory is used by the TSDI.

The tdiDriverRegister() routine returns a ***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 ***driverID.*** The PBX Driver is not allowed to interchange messages from one TSDI registration to another. The tdiDriverRegister() routine is a blocking function that will return to the PBX Driver after the Tserver NLM has initiated the Service Advertising procedures.

4.1.1.2. TSDI Memory Allocation

The Tserver 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 Tserver is not charged against the total TSDI space.

4.1.1.4. Driver Registration Security Level

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

TDI_CSTA_SECURITY	NetWare Login and Password will be validated on the **acsOpenStream()** request. Entry in the Tserver'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	NetWare Login and Password will be validated on the **acsOpenStream()** request. Entry in the Tserver'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.

4.1.2. TSDI Version Control

The *version* field in the tdiDriverRegister() function is used to enforce version control of the TSDI. Currently only one version, **TSDI_VERSION,** of the TSDI exists. The ***version*** field must be set to **TSDI_VERSION.**

4.1.3. 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 encoded in message buffers (see section {4.2.3} below) 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 NetWare OS by the PBX Driver (via the NetWare free() routine). The message buffer must be returned back to the Telephony Services Driver Interface instance. Recall if a Driver wishes to, it is allowed to register CSTA or OA&M services multiple times.

The tdiFreeBuffer() routine can be used to return the message buffer back to the TSDI, or the message buffer may be populated with a request or response message and sent back across the Telephony Services Driver interface to the (CSTA or OA&M) client (see section {4.1.3}). The tdiFreeBuffer() routine is a non-blocking routine and the parameters include a pointer to the buffer that will be released and a ***driverID.*** The same ***driverID*** that was used in the *tdiReceiveFromTserver()* routine must be applied to the tdiFreeBuffer() routine.

4.1.4. 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 encoded in message buffers (see section {4.2.3} below) 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 above in section {3.1.2}), 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 on the interface, 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 ***plus*** **TDI_HDR_SIZE** (currently 12 bytes) bytes needed for a TSDI header which is used to track this piece of TSDI

memory.
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 the it ***waits*** on a NetWare Local Semaphore that is only signalled when the Tserver sends the Driver a message. The tdiSendToTserver() routine, however, is non-blocking because it ***signals*** a different NetWare Local Semaphore that the Tserver waits on for receiving messages from the Driver.

4.1.5. 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 which by default is placed in the error log file and sent to the Tserver's OA&M client. In the current version of the TSDI, no other recovery action is taken.

4.1.6. Unregistering the Driver

The PBX Driver must unregister before unloading, or any time it needs to break the Telephony Services Driver Interface connection. The tdiDriverUnregister() routine requires a single parameter, the ***driverID*** returned to the PBX Driver by the tdiDriverRegister() routine. This routine will block while waiting to send all messages sent from the Driver to the TServer via the tdiSendToTserver() and clear all (NetWare) Threads from this Telephony Services Driver Interface. The resources allocated for this interface will be released back to the NetWare OS, and control will be returned to the PBX Driver. No resources allocated for this interface should be accessed by the PBX Driver after the tdiDriverUnregister() routine completes successfully.

4.1.7. Telephony Services Driver Interface Monitoring

The Telephony Services Driver Interface 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** Tserver NLMs, and the tdiQueueSize() routine provides the count of messages queued to the Tserver and to the PBX Driver for this Telephony Services Driver Interface. 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 Tserver provides no form of flow

control across the TSDI. There is a form of flow control in that fact that once all the memory in a given TSDI instance is allocated to TSDI buffers via tdiAllocBuffer() funcation calls, no more TSDI buffers could be allocated (by the Tserver or Driver) to be passed through the TSDI.

4.1.7. Telephony Server Flow Control Of TSDI Messages

The Telephony Server will flow control messages ***sent to*** to the Driver via the ***tdiSendToDriver()*** routine as follows: When the amount of TSDI memory allocated reaches the ***hiwater_mark*** as defined in the ***buffer_descriptor*** which is part of the ***tdiDriverRegister()*** call, the Tserver will reject all new requests (including ACSOpenStream()) with an error code of **TSERVER_DRIVER_CONGESTION.** 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 is has hit the ***max_bytes*** level, then the Tserver will drop an ACS Stream when a new requeust comes in with an error code of **TSERVER_NO_TDI_BUFFERS**. If the Server is out of short term memory, then the the Tserver will drop an ACS Stream when a new requeust comes in with an error code of **TSERVER_NO_MEMORY.**

This means the driver should chose 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.

4.2. PBX Driver to Client Interface

4.2.1. 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 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 section 10) will attempt to open an *OA&M stream* to a PBX Driver that had previously registered for OA&M services. The Tserver NLM will advertise 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 to tdiDriverRegister() to create the advertised name. The name will look as follows:

Total length of advertised name is 48 Bytes   (including a NULL terminator and “#” delimiting each field)

6 Bytes	10 Bytes	5 Bytes	<= 23 Bytes
vendor_name	service_name	service_type	Server Name

In the following examples, assume that the name of the server on which the Tserver NLM is running is DAGOTTO.

Description	**Example Name**
Tserver OA&M	NOVELL#TSRV_OAM#OAM#DAGOTTO
AT&T CSTA Simulator	ATT#CSTASERV#CSTA#DAGOTTO

The Tserver NLM also provides routing and security services for messages exchanged between the PBX Driver and its Clients along these CSTA or OA&M Streams.

4.2.2. 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 Tserver for a specific service. Two service advertising types are allocated from Novell®, a CSTA service class and an OA&M service class. The PBX Driver must specify one of these service classes during driver registration, and the combination of the ***vendor_name***, ***service_name*** and ***service_type*** parameters provided by the Driver is used to create a unique name to advertise for the PBX Driver service on the internetwork. The tdiDriverRegister() routine.will guarantee that no two Driver registrations result in the same advertised name.
µ §

Figure 3: 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 class definition in section {4.2.2} below) and sent over the Telephony Services Driver Interface to the PBX Driver (see Figure 3). 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, and a CSTA message must be sent back across the interface by the Driver 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 (see section {4.3} below) until the stream is disconnected by the PBX Driver (via the ACSAbortStream) or the Client (via the cstaCloseStream() or acsAbortStream() function call).

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. The *Client Session ID* 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 Telephony Services Driver Interface for this Client session. A Driver Control block (see {4.2.3} below) 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

4.2.3. 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. The Message Block is variable in length, and the format of the Message Block 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 Block. The Private Block is an a optional portion of the messages exchanged between the PBX Driver and the Client over the TSDI. The Private Block is variable in length, and the format of the Private Block is defined before hand between the client and the Driver.

The rest of the fields in the Driver Control block are defined in the following sections.

.

µ §

Figure 4: Message Format

4.2.3.1. The Driver Control Block

The first bytes of the messages exchanged across the Telephony Services Driver Interface must be

the Driver Control Block. The Driver Control Block (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 Driver Control 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. The DC is a C structure of fixed size that is located starting 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 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 in the message buffer as defined by the message and private offset fields.
The DC, 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 allocate 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.

4.2.3.2. The Driver Control Block Field Definition

```
/* Driver Control Block Structure */
typedef struct {
  unsigned short       messageOffset;
  unsigned short       messageLength;
  unsigned short       privateOffset;
  unsigned short       privateLength;
  InvokeID_t         invokeID;
  CSTAMonitorCrossRefID_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 values 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 unsolicted 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 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 in the message buffer. The ***messageType*** field defines each message within a ***messageClass***. See section {4.3} below for the ***messageType*** definitions for CSTA messages, and section {4.4} below 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 **messsageClass** that a driver could receive.

MESSAGE CLASS	**DESCRIPTION**
ACSREQUEST	*ACS request messages sent by a client to the driver (e.g acsOpenStream, etc).* *The **API calls** in chapter 4 of the **[TSAPI]** define the set of messages that make up this class.*
ACSUNSOLICITED	*ACS messages generated asynchronously by the Driver or Tserver to be sent to a client (e.g ACSUniversalFailureEvent,etc)* *See the structure **ACSUnsolicitedEvent** in **acs.h** for a **complete** enumeration of the messages that make up this class.*

ACSCONFIRMATION	*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 **ACSConfirmationEvent** in **acs.h** for a **complete** enumeration of the messages that make up this class.*
CSTAREQUEST	*CSTA request messages sent by a client to the driver (e.g cstaMakeCall, etc), or CSTA request message sent by the driver to a client (e.g, CSTARouteRequestEvent, etc).* *See the structure **CSTARequestEvent** in **csta.h** for a **partial** enumeration of the messages that make up this class. The rest of the messages that make up this class are defined by the* **API calls** *in **[TSAPI].***
CSTAUNSOLICITED	*CSTA messages generated asynchronously by the Driver to be sent to a client (e.g CSTAEstablishedEvent , CSTADeliveredEvent,etc)* *See the structure **CSTAUnsolicitedEvent** in **csta.h** for a **partial** enumeration of the messages that make up this class.*
CSTACONFIRMATION	*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).* *See the structure **CSTAConfirmationEvent** in **csta.h** for a **complete** enumeration of the messages that make up this class.*
CSTAEVENTREPORT	*CSTA message sent by the Driver to a client which reports on some CSTA event, but which does not require the client to send a confirmation event in response.* *See the structure **CSTAEventReport** in **csta.h** for a **complete** enumeration of the messages that make up this class.*
TDRVRREQUEST	*Driver OA&M request messages sent by a client to the driver (e.g TSRVDriverOAMReq, etc).* *The **API calls** in chapter 10,* OA&M API Manual Pages *of the this document define the set of messages that make*

	up this class.
TDRVRUNSOLICITED	*Driver OA&M messages generated asynchronously by the Driver to be sent to a client (e.g TSRVDriverOAMEvent,etc)* *See the structure **TSRVDriverUnsolicitedEvent** in **tdrvr.h** for a **complete** enumeration of the messages that make up this class.*
TDRVRCONFIRMATION	*Driver OA&M confirmation messages sent by the Driver to a client in response to a previous Driver OA&M request message (e.g TSRVDriverOAMConfEvent,etc).* *See the structure **TSRVDriverConfirmationEvent** in **tdrvr.h** for a **complete** enumeration of the messages that make up this class.*

4.2.3.3. DC Block / Message Class Mapping

Only certain fields in the DC Block are valid for each of the different ***messageClass*** defined in the table above for messages sent from the Tserver to the Driver or from the Driver to the Tserver (and then on to clients). The tables below detailed which fields a Driver should expect are valid in messages sent by the Tserver and which fields a Driver must fill in for messages sent by a Driver.

4.2.3.3.1. Tserver to Driver Messages

The tables below indicate with a **X** which fields are set by the Tserver 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.

DC Block Field	ACSREQUEST	CSTAREQUEST	CSTACONFIRMATION Note: the only message sent from the Tserver to the Driver of this class is a CSTARouteSelect() message.	TDRVRREQUEST
messageOffset	X	X	X	X

messageLength	X	X	X	X
privateOffset	X	X	X	X
privateLength	X	X	X	X
invokeID	X	X	X	X
monitorCrossRefID				
sessionID	X	X	X	X
messageClass	X	X	X	X
messageType	X	X	X	X
class_of_service	X	X		

The following class of messages are never sent from the Tserver to a Driver: ACSUNSOLICITED, ACSCONFIRMATION, CSTAUNSOLICITED, CSTAEVENTREPORT, TDRVRUNSOLICITED and TDRVRCONFIRMATION.

4.2.3.3.2. Driver to Tserver Messages

The tables below indicate with a **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.

DC Block Field	ACSCONFIRMATION	ACSUNSOLICITED	TDRVRCONFIRMATION	TDRVRUNSOLICITED
messageOffset	X	X	X	X
messageLength	X	X	X	X
privateOffset	X	X	X	X
privateLength	X	X	X	X
invokeID	X		X	
monitorCrossRefID				

sessionID	X	X	X	X
messageClass	X	X	X	X
messageType	X	X	X	X
class_of_service	X	X		

DC Block Field	CSTAREQUEST	CSTACONFIRMATION	CSTAUNSOLICITED	CSTAEVENTREPORT
messageOffset	X	X	X	X
messageLength	X	X	X	X
privateOffset	X	X	X	X
privateLength	X	X	X	X
invokeID	X	X		

monitorCrossRefID			X (Validate in all unsolicited messages except cstaPrivateStatusEvent and cstaSysStatReqEvent)	
sessionID	X	X	X	X
messageClass	X	X	X	X
messageType	X	X	X	X
class_of_service				

The following class of messages are never sent from the Driver to a Tserver: ACSREQUEST, TDRVREQUEST.

4.2.3.3. 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 a ACS stream are given in section 4.3 {ACS Messaging Interface}. Details on how to handle the request-response protocol for CSTA messages is given in section 4.4 {CSTA Messaging Interface} and details on handling the request-response protocol for Driver OA&M messages is given in section 4.5 {OA&M Messaging Interface}.

4.2.3.3. TSDI Session ID to ACS Handle Mapping

The Session ID in the DC block is used by the Tserver to route Driver messages (confirmation or unsolicited) back to client applications. The Session ID is the actual SPX connection ID (server side) back to the client workstation. The ACS Handle which 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 (Windows DLL or NetWare TLIB). The client API Library maintains a mapping between the ACS Handle the SPX connection ID (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.

4.2.3.3. 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 TDI Driver Registration. A TDI 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.

4.2.3.3. Scope of Invoke IDs

The Tserver gaurantes that invokeIDs passed to the driver will be unique within an ACS stream. The Tserver actually saves the client generated invokeID (either application generated for 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 invoke ID in the DC Block with the saved invoke ID before the event is sent back to the application. A driver can use exported TDI routine, tdiMapInvokeID() to determine the acutally client generated invokeID. A driver should always use the client invokeID in any trace or debugging information it generates related to a stream.

4.3. ACS Messaging Interface

This section illustrates 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 (e.g. 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, and see Section 9 for a definition of the associated C structures and message types.

ACS request, response and event messages transported across the Telephony Services Driver Interface 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 (see Section 9). 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 in order 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-Section 9), the message class and type of the C structure, the associated C structure name of the confirmation event.

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

acs<function name>()	acs API function call name
ACS<function name>Event	ACS Event (request, confirmation or unsolicited)
<function name>_t	C structure name corresponding to a ACS API call, confirmation event or unsolicited event.
<define name>	message type name

4.3.1. Application Control Services

Application Control Services

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Event Structure**
acsOpenStream()	ACSOpenStream_t	ACSREQUEST	ACS_OPEN_STREAM	ACSOpenStreamConfEvent_t
ACSOpenStreamConfEvent	ACSOpenStreamConfEvent_t	ACSCONFIRMATION	ACS_OPEN_STREAM_CONF	NA
acsCloseStream()	ACSCloseStream_t	ACSREQUEST	ACS_CLOSE_STREAM	ACSCloseStreamConfEvent_t

ACSCloseStreamConfEvent	ACSCloseStreamConfEvent_t	ACSCONFIRMATION	ACS_CLOSE_STREAM_CONF	NA
acsAbortStream()	ACSAbortStream_t	ACSREQUEST	ACS_ABORT_STREAM	NA
ACSUniversalFailureEvent	ACSUniversalFailureEvent_t	ACSUNSOLICITED	ACS_UNIVERSAL_FAILURE	NA
ACSUniversalFailureConfEvent	ACSUniversalFailureConfEvent_t	ACSCONFIRMATION	ACS_UNIVERSAL_FAILURE_CONF	NA

4.3.2. Processing ACS Control Messages

4.3.2.1. Processing an acsOpenStream Message

The first thing that the PBX Driver will receive for a new Client session is the (CSTA or OA&M) "open request" message. The open request is sent to the driver so that the driver has an oppurtunity 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 (i.e. the CTI-LINK is down).
NOTE: At this time there are no standard requirements on how drivers should the interaction between the CTI-LINK being down and accepting new open stream requests or tearing down existing ACS streams.
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 **message type** defines which type of message within the indicated **messageClass** this message is. 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 sections 4.3, 4.4 and 4.5 respectively. The **messageClass** and **messageType** indicate what type of message (i.e. C-structure) is in the Driver Control block. The **invoke ID** is used to implement the request-response protocol from the Client application to the PBX Driver. The **invoke ID** is generated by the (acsOpenStream()) API function call (it is also allowable for the application to specify the **invoke ID).** This **invoke ID** must be returned in the "open response" message so that the application can

match it to a previous “open request”.

If the PBX Driver is intent on honoring the “open request”, it must generate an **ACSOpenStreamConfEvent** message using the **sessionID**, and the **invoke ID** from the acsOpenStream()message. The PBX Driver must set the **messageClass** and **message** type to the values defined for an **ACSOpenStreamConfEvent** which would be **ACSCONFIRMATION** and **ACS_OPEN_STREAM_CONF** respectively.

If the PBX Driver is not intent on honoring the “open request”, it must generate an **ACSUniversalFailureConfEvent** message using the **sessionID**, and the **invoke ID** from the acsOpenStream()message. The PBX Driver must set the **messageClass** and **message type** 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.

4.3.2.2. 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 a 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 **message class** will identify the message as a CSTA or OA&M message.

4.3.2.2.1. Processing an acsCloseStream()

In the case of a acsCloseStream() the **messageClass** field will be **ACSREQUEST** and the **messageType** field will be **ACS_CLOSE_STREAM**. The **invoke ID** is generated by the acsCloseStream() API function call. This **invoke ID** 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 **invoke ID** from the acsCloseStream()message, with **messageClass** and **messageType** set appropriately for an **ACSCloseConfirmationEvent** message, i.e **messageClass** set to **ACSCONFIRMATION** and **messageType** set to **ACS_CLOSE_STREAM_CONF.** Unlike the situation on an “open request” where the Driver can chose 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.

4.3.2.2.2. Processing an acs AbortStream()

In the case of a acsAbortStream() the **messageClass** field will be **ACSREQUEST** and the **messageType** field will be **ACS_ABORT_STREAM**. There is no **invoke ID** on this message since no reply from the Driver is required. The Driver must always accept a “abort stream” request and clean up any resources associated with that stream.

4.3.2.3. Asychronously Closing a Stream from the Driver

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

4.3.2.4. When To Use ACSUniversalFailureConfEvent

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

An **ACSUniversalFailureConfEvent** can be also be use to NACK a CSTA or OA&M request such as cstaMakeCall(), **but only** if the reason for the NACK is related to the 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].**

4.3.2.5. When To Use ACSUniversalFailureEvent

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

4.3.2.6. Failure Codes To Use in ACSUniversalFailure Type Messages

Both **ACSUniversalFailureConfEvent** and **ACSUniversalFailureEvent** share the same set of error codes, and the Tserver and Driver must also share this set of error codes for use in these messages. Error code values from 1-999 are reserved for Tserver and values from 1000 and above are available for Driver use. Currently, a small number of generic Driver error codes are defined in **Chapter 4, ACSUniversalFailureEvent Section of [TSAPI].**

The relevant portions of this chapter are reproduced below.

```
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_RESOURCE_LIMITATION = 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. This type includes one of the following specific error values:

Driver Duplicate ACSHandle

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

Driver Invalid 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 (i.e. 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 offline.

4.4. 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 (see Section 9). 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) in order to access the information contained in the CSTA message.

There are three categories of CSTA API calls and Events:

- Switching Function Services
- Status Reporting Services
- CSTA Computing Function 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-Section 9), the message class and type of the C structure, the associated C structure name of the confirmation event.

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

csta<function name>	csta API function call name
CSTA<function name>Event	CSTA Event (request, confirmation or unsolicited)
<function name>_t	C structure name corresponding to a CSTA API call, confirmation event or unsolicited event.
<define name>	message type name

4.4.1. 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 (CSTA or OA&M) API function calls 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 (CSTA, ACS or OA&M) API function call, and the **invoke ID** 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 sections {4.3 , 4.4 and 4.5} below). 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 (CSTA or OA&M) client. The PBX Driver must include the **sessionID**, the **messageClass**, the **messageType** in event message sent to the Client application. An **invoke ID** is not used for events that are originated by the PBX Driver.
Requests may also be generated by the PBX Driver for the (CSTA or OA&M) client. The PBX Driver must include the **sessionID**, the **messageClass**, the **messageType** in the request sent to the Client application. An **invoke ID** is used for requests (except for Routing Reqeusts- see 4.4.7) that are originated by the PBX Driver. These request messages are mapped into events for the Client at the (CSTA or OA&M) API. The Client must call an (CSTA or OA&M) API function 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 .**invoke ID.**

4.4.2. Processing CSTA Messages

A Driver must respond in one of three ways for each **CSTAREQUEST** message a received:
Send back the appropriate **CSTACONFIRMATION** message indicating the the requested operation has been initiated. Refer to **[ECMA-179]** for a description/definition of what each confirmation event means.
Send back a **UniversalFailureConfEvent** messages 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 a **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.

4.4.3. CSTA Control Services Functions

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

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Event Structure**
cstaGetAPICaps()	CSTAGetAPICaps_t	CSTAREQUEST	CSTA_GETAPI_CAPS	CSTAGetAPICapsConfEvent_t
CSTAGetAPICapsConfEvent	CSTAGetAPICapsConfEvent_t	CSTACONFIRMATION	CSTA_GETAPI_CAPS_CONF	NA

4.4.4. 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, i.e. every API call in the **[TSAPI]** is listed in this document to make clear which of these messages the Driver must deal with and which the Tserver deals with.

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Event Structure**
cstaGetDeviceList()	CSTAGetDeviceList_t	CSTAREQUEST	CSTA_GET_DEVICE_LIST	CSTAGetDeviceListConfEvent_t
CSTAGetDeviceListConfEvent	CSTAGetDeviceListConfEvent_t	CSTACONFIRMATION	CSTA_GET_DEVICE_LIST_CONF	NA
cstaQueryCallMonitor()	CSTAQueryCallMonitor_t	CSTAREQUEST	CSTA_QUERY_CALL_MONITOR	CSTAQueryCallMonitorConfEvent_t
CSTAQueryCallMonitorConfEvent	CSTAQueryCallMonitorConfEvent_t	CSTACONFIRMATION	CSTA_QUERY_CALL_MONITOR_CONF	NA

4.4.5. 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 Telephony Services Driver Interface. Switching function services are Telephony Services which 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, and see Section 9 for a definition of the related C structures and message types.

4.4.5.1. Basic Call Control Services

This section defines Telephony Services which deal with basic call control for the desktop or call center environments. These functions provide services which 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/de-activate features and capabilities supported by the switch or the server.

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Event Structure**
UniversalFailureConfEvent	UniversalFailureConfEvent_t	CSTACONFIRMATION	CSTA_UNIVERSAL_FAILURE_CONF	NA
cstaAlternateCall()	CSTAAlternateCall_t	CSTAREQUEST	CSTA_ALTERNATE_CALL	CSTAAlternateCallConfEvent_t
CSTAAlternateCallConfEvent	CSTAAlternateCallConfEvent_t	CSTACONFIRMATION	CSTA_ALTERNATE_CALL_CONF	NA
cstaAnswerCall()	CSTAAnswerCall_t	CSTAREQUEST	CSTA_ANSWER_CALL	CSTAAnswerCallConfEvent_t
CSTAAnswerCallConfEvent	CSTAAnswerCallConfEvent_t	CSTACONFIRMATION	CSTA_ANSWER_CALL_CONF	NA
cstaCallCompletion()	CSTACallCompletion_t	CSTAREQUEST	CSTA_CALL_COMPLETION	CSTACallCompletionConfEvent_t
CSTACallCompletionConfEvent	CSTACallCompletionConfEvent_t	CSTACONFIRMATION	CSTA_CALL_COMPLETION_CONF	NA
cstaClearCall()	CSTAClearCall_t	CSTAREQUEST	CSTA_CLEAR_CALL	CSTAClearCallConfEvent_t
CSTAClearCallConfEvent	CSTAClearCallConfEvent_t	CSTACONFIRMATION	CSTA_CLEAR_CALL_CONF	NA
cstaClearConnection()	CSTAClearConnection_t	CSTAREQUEST	CSTA_CLEAR_CONNECTION	CSTAClearConnectionConfEvent_t
CSTAClearConnectionConfEvent	CSTAClearConnectionConfEvent_t	CSTACONFIRMATION	CSTA_CLEAR_CONNECTION_CONF	NA

cstaConferenceCall()	CSTAConferenceCall_t	CSTAREQUEST	CSTA_CONFERENCE_CALL	CSTAConferenceCallConfEvent_t
CSTAConferenceCallConfEvent	CSTAConferenceCallConfEvent_t	CSTACONFIRMATION	CSTA_CONFERENCE_CALL_CONF	NA
cstaConsultationCall()	CSTAConsultationCall_t	CSTAREQUEST	CSTA_CONSULTATION_CALL	CSTAConsultationCallConfEvent_t
CSTAConsultationCallConfEvent	CSTAConsultationCallConfEvent_t	CSTACONFIRMATION	CSTA_CONSULTATION_CALL_CONF	NA
cstaDeflectCall()	CSTADeflectCall_t	CSTAREQUEST	CSTA_DEFLECT_CALL	CSTADeflectCallConfEvent_t
CSTADeflectCallConfEvent	CSTADeflectCallConfEvent_t	CSTACONFIRMATION	CSTA_DEFLECT_CALL_CONF	NA
cstaGroupPickupCall()	CSTAGroupPickupCall_t	CSTAREQUEST	CSTA_GROUP_PICKUP_CALL	CSTAGroupPickupCallConfEvent_t
CSTAGroupPickupCallConfEvent	CSTAGroupPickupCallConfEvent_t	CSTACONFIRMATION	CSTA_GROUP_PICKUP_CALL_CONF	NA
cstaHoldCall()	CSTAHoldCall_t	CSTAREQUEST	CSTA_HOLD_CALL	CSTAHoldCallConfEvent_t
CSTAHoldCallConfEvent	CSTAHoldCallConfEvent_t	CSTACONFIRMATION	CSTA_HOLD_CALL_CONF	NA
cstaMakeCall()	CSTAMakeCall_t	CSTAREQUEST	CSTA_MAKE_CALL	CSTAMakeCallConfEvent_t
CSTAMakeCallConfEvent	CSTAMakeCallConfEvent_t	CSTACONFIRMATION	CSTA_MAKE_CALL_CONF	NA
cstaMakePredictiveCall()	CSTAMakePredictiveCall_t	CSTAREQUEST	CSTA_MAKE_PREDICTIVE_CALL	CSTAMakePredictiveCallConfEvent_t
CSTAMakePredictiveCallConfEvent	CSTAMakePredictiveCallConfEvent_t	CSTACONFIRMATION	CSTA_MAKE_PREDICTIVE_CALL_CONF	NA

cstaPickupCall()	CSTAPickupCall_t	CSTAREQUEST	CSTA_PICKUP_CALL	CSTAPickupCallConfEvent_t
CSTAPickupCallConfEvent	CSTAPickupCallConfEvent_t	CSTACONFIRMATION	CSTA_PICKUP_CALL_CONF	NA
cstaReconnectCall()	CSTAReconnectCall_t	CSTAREQUEST	CSTA_RECONNECT_CALL	CSTAReconnectCallConfEvent_t
CSTAReconnectCallConfEvent	CSTAReconnectCallConfEvent_t	CSTACONFIRMATION	CSTA_RECONNECT_CALL_CONF	NA
cstaRetrieveCall()	CSTARetrieveCall_t	CSTAREQUEST	CSTA_RETRIEVE_CALL	CSTARetrieveCallConfEvent_t
CSTARetrieveCallConfEvent	CSTARetrieveCallConfEvent_t	CSTACONFIRMATION	CSTA_RETRIEVE_CALL_CONF	NA
cstaTransferCall()	CSTATransferCall_t	CSTAREQUEST	CSTA_TRANSFER_CALL	CSTATransferCallConfEvent_t
CSTATransferCallConfEvent	CSTATransferCallConfEvent_t	CSTACONFIRMATION	CSTA_TRANSFER_CALL_CONF	NA

4.4.6. 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 Telephony Services Driver Interface. 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 (e.g. a desktop telephone). See **[ECMA-179]** for more details on the status reporting services, see **TSAPI** for more information on the associated CSTA API C Language function calls and events, and see Section 9 for a definition of the related C structures and message types.

4.4.7.

µCSTA 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 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.

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Event Structure**
cstaSnapshotCallReq( )	CSTASnapshotCall_t	CSTAREQUEST	CSTA_SNAPSHOT_CALL	CSTASnapshotCallConfEvent_t
CSTASnapshotCallConfEvent	CSTASnapshotCallConfEvent_t	CSTACONFIRMATION	CSTA_SNAPSHOT_CALL_CONF	NA
cstaSnapshotDeviceReq( )	CSTASnapshotDevice_t	CSTAREQUEST	CSTA_SNAPSHOT_DEVICE	CSTASnapshotDeviceConfEvent_t
CSTASnapshotDeviceConfEvent	CSTASnapshotDeviceConfEvent_t	CSTACONFIRMATION	CSTA_SNAPSHOT_DEVICE_CONF	NA

4.4.8. 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 Telephony Services Driver Interface. 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, see **TSAPI** for more information on the associated CSTA API C Language function calls and events, and see Section 9 for a definition of the related C structures and message types.

µ

4.4.8.1. Routing Registration Functions and Events

This section describe the service requests and events which are used by an application to register

with the Telephony Server as a call routing server for a specific routing device.

API Call or ACS Event Name	C Language Structure Name	Message Class	MessageType	C Language Confirmation Event Structure
cstaRouteRegisterReq( )	CSTARouteRegisterReq_t	CSTAREQUEST	CSTA_ROUTE_REGISTER_REQ	CSTARouteRegisterReqConfEvent_t
CSTARouteRegisterReqConfEvent	CSTARouteRegisterReqConfEvent_t	CSTACONFIRMATION	CSTA_ROUTE_REGISTER_REQ_CONF	NA
cstaRouteRegisterCancel( )	CSTARouteRegisterCancel_t	CSTAREQUEST	CSTA_ROUTE_REGISTER_CANCEL	CSTARouteRegisterCancelConfEvent_t
CSTARouteRegisterCancelConfEvent	CSTARouteRegisterCancelConfEvent_t	CSTACONFIRMATION	CSTA_ROUTE_REGISTER_CANCEL_CONF	NA
CSTARouteRegisterAbortEvent	CSTARouteRegisterAbortEvent_t	CSTAEVENTREPORT	CSTA_ROUTE_REGISTER_ABORT	NA

4.4.8.2. Routing Functions and Events

This section defines the CSTA call routing services which can be utilized for application-based call routing within the switching domain. Calls which are routed using these services are queued at the routing device until the application provides a destination for the call or a time-out 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 and is illustrated in Figure 8-1.

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*" for more details on registering as a routing server). A **CSTARouteRequestEvent** will be sent to the application for each call which requires a routing destination from the routing server, i.e. the application. The route request response is issued by the application using the **cstaRouteSelect( )** function which provides the switch with the appropriate destination for the call (e.g. 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 **cstaRouteSelect( )** function. The application should check the **CSTARouteEndEvent** and/or the **CSTARouteUsedEvent** to insure 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 **CSTAReRouteEvent**. See Figure 8-1 for a typical sequence of these events and services requests.

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.

API Call or ACS Event Name	C Language Structure Name	Message Class	MessageType	C Language Confirmation Routine Used by Client
CSTARouteRequestEvent	CSTARouteRequestEvent_t	CSTAREQUEST	CSTA_ROUTE_REQUEST	cstaRouteSelect( )
CSTAReRouteRequestEvent	CSTAReRouteRequestEvent_t	CSTAREQUEST	CSTA_RE_ROUTE_REQUEST	cstaRouteSelect( )

cstaRouteSelect( )	CSTARouteSelectRequest_t	CSTACONFIRMATION	CSTA_ROUTE_SELECT_REQUEST	NA
CSTARouteUsedEvent	CSTARouteUsedEvent_t	CSTAEVENTREPORT	CSTA_ROUTE_USED	NA
CSTARouteEndEvent	CSTARouteEndEvent_t	CSTAEVENTREPORT	CSTA_ROUTE_END	NA
cstaRouteEnd( )	CSTARouteEndRequest_t	CSTAREQUEST*	CSTA_ROUTE_END_REQUEST	NA (No confirmation event required for this request message)

4.4.9.

µCSTA Escape/Maintenance Services

There are two different types of maintenance services defined within the CSTA standard :

- the device status maintenance events which provide status information for device objects and
- bi-directional system status maintenance services which provides 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 (e.g. **cstaMonitorDevice( )** or **cstaDevSnapshotReq( )**) 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 which is being used to monitor the device, i.e. the ***monitorCrossRefID***. The Driver must enforce this limitation.

4.4.9.1. Escape Services : Application as Client

This section defines escape services for cases where the application is the service requester in the client/server relationship (see Figure 9-1). 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.

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Event Structure**
cstaEscapeService( )	CSTAEscapeSvc_t	CSTAREQUEST	CSTA_ESCAPE_SVC	CSTAEscapeSvcConfEvent_t
CSTAEscapeSvcConfEvent	CSTAEscapeSvcConfEvent_t	CSTACONFIRMATION	CSTA_ESCAPE_SVC_CONF	NA
CSTAPrivateEvent	CSTAPrivateEvent_t	CSTAEVENTREPORT	CSTA_PRIVATE	NA
CSTAPrivateStatusEvent	CSTAPrivateStatusEvent_t	CSTAUNSOLICITED	CSTA_PRIVATE_STATUS	NA

4.4.9.2. 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.

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Event Structure**
CSTAEscapeSvcReqEvent	CSTAEscapeSvcReqEvent_t	CSTAREQUEST	CSTA_ESCAPE_SVC_REQ	CSTAEscapeSvcConfEvent_t
cstaEscapeServiceConf( )	CSTAEscapeSvcReqConf_t	CSTAREQUEST	CSTA_ESCAPE_SVC_REQ_CONF	NA(No confirmation event required for this request message)
cstaSendPrivateEvent( )	CSTASendPrivateEvent_t	CSTAREQUEST	CSTA_SEND_PRIVATE	NA(No confirmation event required for this request message)

4.4.9.3. Maintenance Services

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

4.4.9.3.1. 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 (e.g. an active ***monitorCrossRefID)***, in order to receive this event. These events are unidirectional and always originate in the driver/switch domain.

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Event Structure**
CSTABackInServiceEvent	CSTABackInServiceEvent	CSTAUNSOLICITED	CSTA_BACK_IN_SERVICE	NA
CSTAOutOfServiceEvent	CSTAOutOfServiceEvent_t	CSTAUNSOLICITED	CSTA_OUT_OF_SERVICE	NA

4.4.9.3.2. System Status - Application as the Client

This section defines the services which 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.

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Event Structure**
cstaSysStatReq( )	CSTAReqSysStat_t	CSTAREQUEST	CSTA_REQ_SYS_STAT	CSTASysStatReqConfEvent_t
CSTASysStatReqConfEvent	CSTASysStatReqConfEvent_t	CSTACONFIRMATION	CSTA_SYS_STAT_REQ_CONF	NA

cstaSysStatStart( )	CSTASysStatStart_t	CSTAREQUEST	CSTA_START_SYS_STAT	CSTASysStatStartConfEvent_t
CSTASysStatStartConfEvent	CSTASysStatStartConfEvent_t	CSTACONFIRMATION	CSTA_SYS_STAT_START_CONF	NA
cstaSysStatStop( )	CSTASysStatStop_t	CSTAREQUEST	CSTA_SYS_STAT_STOP	CSTASysStatStopConfEvent_t
CSTASysStatStopConfEvent	CSTASysStatStopConfEvent_t	CSTACONFIRMATION	CSTA_SYS_STAT_STOP_CONF	NA
cstaChangeSysStatFilter( )	CSTAChangeSysStatFilter_t	CSTAREQUEST	CSTA_CHANGE_SYS_STAT_FILTER	CSTAChangeSysStatFilterConfEvent_t
CSTAChangeSysStatFilterConfEvent	CSTAChangeSysStatFilterConfEvent_t	CSTACONFIRMATION	CSTA_CHANGE_SYS_STAT_FILTER_CONF	NA
CSTASysStatEvent	CSTASysStatEvent_t	CSTAEVENTREPORT	CSTA_SYS_STAT	NA
CSTASysStatEndedEvent	CSTASysStatEndedEvent_t	CSTAEVENTREPORT	CSTA_SYS_STAT_ENDED	NA

4.4.9.3.3. System Status : Driver/Switch as the Client

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

API Call or ACS Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**C Language Confirmation Routine Used by Client**
CSTASysStatReqEvent_t	CSTASysStatReqEvent_t	CSTAUNSOLICITED	CSTA_SYS_STAT_REQ	cstaSysStatReqConf( )
cstaSysStatReqConf( )	CSTAReqSysStatConf_t;	CSTAREQUEST	CSTA_REQ_SYS_STAT_CONF	NA

cstaSysStatEventSend( )	CSTASysStatEventSend_t	CSTAREQUEST	CSTA_SYS_STAT_EVENT_SEND	NA(No confirmation event required for this request message)

4.5. 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.

4.5.1. 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 Telephony Services Driver Interface. OA&M ICS functions deal with the characteristics of the API interface (e.g. opening and closing the OA&M interface). They provide the ability to open, initialize, close and manage a virtual communication channel (OA&M stream) with any advertised OA&M registered Telephony Server. See Sections 10 and 11 for a specification of the OA&M API C Language function calls and events.

OA&M Interface Control Services

API Call or CSTA Event Name	**C Language Structure Name**	**Message Class**	**MessageType**	**Confirmation Event**
tsrvDriverRequest()	NA	TDRVRREQUEST	TSRV_DRIVEROAM_REQ	TSRVDriverOAMConfEvent

TSRVDriverOAMConfEvent	TSRVDriverOAMConfEvent_t	TDRVRCONFIRMATION	TSRV_DRIVEROAM_CONF	NA
TSRVDriverOAMEvent	TSRVDriverOAMEvent_t	TDRVRUNSOLICITED	TSRV_DRIVEROAM	NA

4.6. 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 interpretted 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 Application.

4.7. Error Log Interface

The Tserver exports a standard function call interface to the PBX Driver so that the Driver can log errors to the Telephony Server log file. When a PBX Driver uses the *tdiLogError()* function, the errors reported by the Driver and the Tserver will have a uniform appearance in the error log. The error log interface will provide 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 non-service 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 of itself is not service-affecting, but indicates a condition that may become a problem (e.g., 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 NLM.

There are three possible destinations for each severity level:

- the Tserver's Error Log File
- the System Console Screen
- the Tserver's OA&M client

These destinations can be set through the Tserver's OA&M client. Because these destinations are set through and the log if viewed through the Tserver OA&M application, and there is no provision for defining a Driver OA&M destination.

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

The manual page for this interface is in Section 12.

5. Compiling and Linking a Driver

The Telephony Server was been compiled using the WATCOM C/C++ 32 bit compiler, version 9.5 with the zp1 option. The zp1 causes the code to be compiled on a single byte ordering. This means all drivers must be compiled with single (one) byte ordering so that the C-structures passed across the TSDI have the same memory layout in the Tserver and the Driver.

The following is a variable from the makef ile used to build the TSRV.NLM which shows the

compiler options used.

p_wcc386opt = -I..\..\..\hdrs -I..\..\hdrs -I..\hdrs -I.\hdrs /w4 /zp1 /3s /zl /od /d2

The TDI library routines defined in section 7 (i.e tdiDriverRegister, tdiAllocBuffer, etc) are export by the TServer NLM using the NetWare export linker directive. A Driver must use the NetWare import directive in the linking phase of building. When the Driver NLM is loaded, these TDI functions will by dynamically linked to the Driver NLM.

6. TSDI Coding Examples

6.1. 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:                        */
/*                          */
/*****************************************************************************/
main()
{

        int     rc;

        /*
         * Register an Unload cleanup function
         */

        signal(myAtUnload, SIGTERM);

        /*
         * Register the driver for CSTA Services
         */
```

```
CstaTDIHandle = tdiDriverRegister("CSTASERV", "driver_name",
                        TDI_ST_CSTA, NULL, "ATT", TSDI_VERSION,
                        TDI_CSTA_SECURITY, NULL);

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

:
:
:

/*
 * Begin Thread that will send the sanity message
 * to the Tserver every minute.
 */
rc = BeginThreadGroup(sanity,NULL,8192,&CstaTDIHandle);

if (rc == EFAILURE)
{
        tdiLogError(DRIVER_NAME, FATAL, ERR_NO + 1, 0,
                "Couldn't begin thread group for sanity timer. Error number = %d",errno);
        return;
}
:
:
:
}
```

```
/*****************************************************************/
/* myAtUnload: Unregisters with the TSDI, cleans up resources    */
/*                                                               */
/*****************************************************************/
myAtUnload()
{
        int       rc;

        /*
         * Unregister the CSTA Driver
         */
        if ((rc = tdiDriverUnregister(CstaTDIHandle)) != TDI_SUCCESS)
        {
                tdiLogError(DRIVER_NAME, FATAL, ERR_NO, 0,
                        "tdiDriverUnRegister failed : %d",rc);
        }

        /*
         * Clean up any other remaining resources
         */

}

/*****************************************************************/
/* This is the sanity thread which sends the sanity message to the  */
/* Tserver once every minute.                    */
/*                                                               */
/*****************************************************************/
void sanity(TDIHandle_t *tdiHandle)
{
char    threadName[30];
```

```
        sprintf(threadName,"DRIVERSANITY");
        RenameThread(GetThreadID(),threadName);

        while(TRUE)
        {
                tdiDriverSanity(tdiHandle);
                delay(60000);
        }
}
```

6.2. Processing an ACSOpenStream() Request

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

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

```
        TDIDriverControlBlock_t         *idc;           /* incoming 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 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 are not supported by the driver an ACSUniversalFailureConfEvent should be returned.

```
char                            *obuffer;       /* outgoing buffer */
ACSOpenStream_t                 *iMsg; /* request msg */
ACSOpenStreamConfEvent_t        *oEvent;        /* conf event */
int                             rc;
TDIBuf_flag_t                   buf_flag;

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

or

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



/* set a ptr to the allocated buffer to fill in
 * the confirmation event or universal failure 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);
```

6.2.1. 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.

```
/*
 * 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);

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

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


/* send the confirmation event to the Tserver */
rc = tdiSendToTserver(tsdiHandle,obuffer,TDI_NORMAL_MESSAGE);
if (rc != TDI_SUCCESS)
{
        tdiLogError(DRIVER_NAME, ERROR, ERR_NO, 0,
                "tdiSendToTserver failed: rc = %d",rc);
}
```

6.2.2. Returning an ACSUniversalFailureConfEvent

Fill in the error code for the failure

```
oEvent->error = yourErrorCode;

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

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

/* send the confirmation event to the Tserver */
rc = tdiSendToTserver(tsdiHandle,obuffer,TDI_NORMAL_MESSAGE);
if (rc != TDI_SUCCESS)
{
        tdiLogError(DRIVER_NAME, ERROR, ERR_NO, 0,
                "tdiSendToTserver failed: rc = %d",rc);
}
```

6.3. Processing an AcsCloseStream() Request

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

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

```
TDIDriverControlBlock_t         *idc;          /* incoming 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 *) ibuffer;


tdiAllocBuffer(tsdiHandle,&obuffer,
                (sizeof(TDIDriverControlBlock_t) +
                 idc->privateLength +
                 sizeof(ACSCloseStreamConfEvent_t)),
                &buf_flag);

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

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

```
rc = tdiSendToTserver(tsdiHandle,obuffer,TDI_NORMAL_MESSAGE);
if (rc != TDI_SUCCESS)
{
        tdiLogError(DRIVER_NAME, ERROR, ERR_NO, 0,
                "tdiSendToTserver failed: rc = %d",rc);
}
```

6.4. 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.

```
typedef struct CSTAConferenceCallConfEvent_t {
  ConnectionID_t newCall;
  ConnectionList_t connList;
} CSTAConferenceCallConfEvent_t;

typedef struct ConnectionList_t {
  int       count;
  Connection_t  FAR *connection;
} ConnectionList_t;

typedef struct Connection_t {
  ConnectionID_t party;
  SubjectDeviceID_t staticDevice;
} Connection_t;

typedef struct ConnectionID_t {
  long      callID;
```

```
  DeviceID_t   deviceID;
  ConnectionID_Device_t devIDType;
} ConnectionID_t;

typedef enum ConnectionID_Device_t {
  STATIC_ID = 0,
  DYNAMIC_ID = 1
} ConnectionID_Device_t;

typedef ExtendedDeviceID_t SubjectDeviceID_t;

typedef enum DeviceIDStatus_t {
  ID_PROVIDED = 0,
  ID_NOT_KNOWN = 1,
  ID_NOT_REQUIRED = 2
} DeviceIDStatus_t;

typedef enum DeviceIDType_t {
  DEVICE_IDENTIFIER = 0,
  IMPLICIT_PUBLIC = 20,
  EXPLICIT_PUBLIC_UNKNOWN = 30,
  EXPLICIT_PUBLIC_INTERNATIONAL = 31,
  EXPLICIT_PUBLIC_NATIONAL = 32,
  EXPLICIT_PUBLIC_NETWORK_SPECIFIC = 33,
  EXPLICIT_PUBLIC_SUBSCRIBER = 34,
  EXPLICIT_PUBLIC_ABBREVIATED = 35,
  IMPLICIT_PRIVATE = 40,
  EXPLICIT_PRIVATE_UNKNOWN = 50,
  EXPLICIT_PRIVATE_LEVEL3_REGIONAL_NUMBER = 51,
  EXPLICIT_PRIVATE_LEVEL2_REGIONAL_NUMBER = 52,
  EXPLICIT_PRIVATE_LEVEL1_REGIONAL_NUMBER = 53,
  EXPLICIT_PRIVATE_PTN_SPECIFIC_NUMBER = 54,
```

```
  EXPLICIT_PRIVATE_LOCAL_NUMBER = 55,
  EXPLICIT_PRIVATE_ABBREVIATED = 56,
  OTHER_PLAN = 60,
  TRUNK_IDENTIFIER = 70,
  TRUNK_GROUP_IDENTIFIER = 71
} DeviceIDType_t;

typedef char      DeviceID_t[64];

typedef struct ExtendedDeviceID_t {
  DeviceID_t   deviceID;
  DeviceIDType_t deviceIDType;
  DeviceIDStatus_t deviceIDStatus;
} ExtendedDeviceID_t;
```

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:

µ §

The call to allocate the buffer may look like this:

```
tdiAllocBuffer(tdiHandle,
                    &buffer,
                    ( sizeof(TDIDriverControlBlock_t) +
```

```
                dc->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.

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

Populate the newCall structure with the relevant information.

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

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

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

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

```
ptr->connList.connection = (Connection_t *)
                ((char *)ptr + sizeof(CSTAConferenceCallConfEvent_t));
```

Index into connList.connection and set the information for each connection

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

6.5. 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. It must be remembered 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 Tserver.

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.

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

PrivateData_t  *privateData;

privateData->length = sizeof(yourOwnDefinedPrivateDataStructure);

dc->privateLength = sizeof(PrivateData_t) - 1 +
                                                privateData->length;



char    pbuf[dc->privateLength];
p = (PrivateData_t *) pbuf;
```

```
/* use the data field as you would any buffer */
p->data ...
```

6.6. 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 to be 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) +
                              idc->privateLength +
                     sizeof(CSTAMonitorConfEvent_t),
                    &buf_flag);

odc  = (TDIDriverControlBlock_t *) obuffer;
```

```
/* 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.
 */

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

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

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

CSTA_DELIVERED event.

```
/* allocate a buffer large enough to
 * hold the DC block, any private data and the
 * unsolicited event.
 */

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

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

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

dc->messageOffset    = sizeof(TDIDriverControlBlock_t);
dc->messageLength            = sizeof(CSTADeliveredEvent_t);
dc->privateOffset    = dc->messageOffset + dc->messageLength;
dc->privateLength    = idc->privateLength;
dc->invokeID         = idc->invokeID;
dc->sessionID        = idc->sessionID;
dc->messageClass             = CSTAUNSOLICITED;
dc->messageType              = CSTA_DELIVERED;
dc->class_of_service  = idc->class_of_service;
```

```
dc->monitorCrossRefID        = msg->monitorCrossRefID;

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

7. Telephony Services Driver Interface Manual Pages tc "Appendix 1: Tserver-Driver Interface Manual Pages"§

The following manual pages describe the function call interface between the Driver NLM and the Tserver NLM.

µ

7.1. tdiDriverRegister ( )

This function allows a Driver NLM (PBX or other) to register itself with the Tserver. It registers its name with the Tserver, specifies some tags that will be maintained by the Tserver NLM for maintenance queries, and requests parameters that specify the memory allocation limits that will be imposed on the Tserver and Driver NLMs for this interface.

Syntax

```
#include <tdi.h>

TDIHandle_t tdiDriverRegister (

    const char      *service_name,              /* INPUT    */
    const char      *driver_name,           /* INPUT    */
    int             service_type,               /* INPUT    */
    int             channel_number,             /* INPUT    */
    const char      *vendor_name,               /* INPUT    */
    const char      *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 (e.g. for every unique **driver_name** all **service_name** must be unqiue.

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).

channel_number

This parameter identifies the interface between the Tserver and the Driver. This parameter is not mandatory and is not used for this release of the Tserver product.

vendor_name

This is the NULL terminated ASCII string that identifies the manufacturer's name of the driver NLM. 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. This parameter is mandatory. The registration will fail if this parameter is set to an invalid version.

driver_security

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

TDI_CSTA_SECURITY	NetWare Login and Password will be validated on the **acsOpenStream()** request. Entry in the Tserver'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	NetWare Login and Password will be validated on the **acsOpenStream()** request. Entry in the Tserver's Security Database must contain this login. This is also

	checked at the time of the **acsOpenStream()** request.
TDI_NO_SECURITY	NetWare Login and Password will be validated on the **acsOpenStream()** request.

buffer_descriptor

This is a pointer to a buffer descriptor structure containing the information about the memory that can be allocated for this Telephony Services Driver interface. A NULL pointer will use the default values listed below for each element of the buffer descriptor.

```
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;
```

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_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 NLM to set up a communication path with the Tserver NLM, identify the name that will be advertised by the Tserver NLM (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 NLMs for this communication path. Both the

Tserver NLM and the Driver NLM must allocate message buffers from the TSDI routines to send a message across this Telephony Services Driver interface.

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 NLMs from the TSDI will include a (*12 byte*) header that will be used to implement the monitoring and queueing of messages. (*The message buffer header should not be accessed by the Driver or the Tserver NLMs, it is used by the TSDI routines.*) This header should 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:

```
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 Tserver and the Driver NLMs for message buffers used on this communication path between the Driver and Tserver. 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 *{0x100000}*.

hiwater_mark

A non-negative integer indicates a high water mark for the memory allocated for this Telephony Services Driver interface. 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 NetWare® OS), and include an indication 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.

.

§

µ

7.2. tdiDriverUnregister ( )

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

Syntax

```
#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 on failure this function returns one of the following (negative) values:

TDI_ERR_BAD_DRVRID	This error indicates that the ***driverID*** specified in the *tdiDriverUnregister()* function is not valid.
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 routine is exported from the Tserver for use by Drivers that import the TSDI routines. It will cause the Tserver to delete the Driver List entry that was created when the driver originally registered with the Tserver via the *tdiDriverRegister()* routine. All memory that was allocated for this Telephony Services Driver Interface will be given back to the NetWare® OS. 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 Telephony Services Driver interface.

§

§

µ

7.4. tdiFreeBuffer ( )

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

Syntax

```
#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 Telephony Services Driver interface.

bufptr

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

Return Values

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

TDI_ERR_BAD_DRVRID

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

TDI_ERR_BAD_BUF

This error indicates that the memory block pointed to by ***bufptr*** is not a currently allocated Telephony Services Driver Interface buffer.

TDI_ERR_NOT_YOUR_BUFFER

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. Note: All TSDI Buffers allocated by a Tserver to be sent across the TSDI to a Driver are allocated with that Drivers 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

The *tdiFreeBuffer()* function returns a buffer to the NetWare® OS that was previously allocated to send a message between the Driver NLM and the Tserver NLM.

Warning

Memory allocated for message buffers via the *tdiAllocBuffer()* routine should not be directly freed back to the NetWare OS by the Tserver or the Driver NLMs; the messages should be released back to the Telephony Services Driver interface via the *tdiFreeBuffer()* routine.

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

Driver NLM Notes

The Driver NLM 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 Telephony Services Driver interface. The Driver NLM is not responsible for the memory resources allocated by the Telephony Services Driver interface since they are allocated from the Novell® NetWare OS in the Tserver context. The Telephony Services Driver interface is responsible for freeing these memory resources. The Driver NLM is responsible for giving the memory resources back to the Telephony Services Driver Interface via the *tdiFreeBuffer()* routine or sending the buffer to the Tserver via the *tdiSendToTserver()* routine.

Tserver NLM Notes

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

§

µ

7.5. tdiSendToTserver( )

This function allows the Driver NLM to send a message buffer to the Tserver NLM. The message will be queued until a corresponding ***tdiReceiveFromDriver()*** routine is called by the Tserver NLM. 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

```
#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 Telephony Services Driver Interface.

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 non-priority 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 on failure this function returns one of the following (negative) values:

TDI_ERR_BAD_DRVRID	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 Telephony Services Driver Interface by the Driver.
TDI_ERR_EINVAL	This error indicates that the ***priority*** parameter contains an invalid value.
TDI_ERR_NOT_YOUR_BUFFER	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 was 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 Drivers 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 NLM to the Tserver NLM. The message will be queued until a corresponding ***tdiReceiveFromDriver()*** routine is called by the Tserver NLM. 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 ***bufptr*** that has been allocated by the Driver NLM via the *tdiAllocBuffer()* routine.

Driver NLM Notes

The Driver NLM 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 send a message to the Tserver via the *tdiSendToTserver()* function call. The Driver NLM must allocate a message buffer via the *tdiAllocBuffer()* function call prior to calling *tdiSendToTserver(),* and the Driver NLM is no longer responsible for that message buffer after *tdiSendToTserver()* has completed successfully.

Tserver NLM Notes

The Tserver NLM is responsible for calling *tdiReceiveFromDriver()* to retrieve messages from its queues in a timely manner, and the Tserver NLM must give the memory buffer back to the Telephony Services Driver Interface via the *tdiFreeBuffer()* routine after the message has been processed, or send this message buffer back to the Driver via the *tdiSendToDriver()* routine.

§

µ

7.6. tdiReceiveFromTserver( )

This function is called by the Driver NLM in order to receive a populated message buffer from the Tserver NLM. 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

```
#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 correctly identify 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 on failure ***bufptr*** is set to NULL and this function returns one of the following (negative) values:

TDI_ERR_BAD_DRVRID	This error indicates that the ***driverID*** specified in the *tdiReceiveFromTserver()* function is not valid. This may indicate that the Driver unregistered during the time this function blocks 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.

Comments

The *tdiReceiveFromTserver()* function receives a message that was sent by the Tserver NLM across the Telephony Services Driver Interface. The buffer will be owned by the Driver NLM until it is sent back to the Tserver NLM 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 NLM to send a message to the Driver.

Driver NLM Notes

The Driver NLM 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 Telephony Services Driver interface. The Driver NLM is responsible for calling *tdiReceiveFromTserver()* to retrieve messages from its queues in a timely manner, and the Driver NLM must give the memory buffer back to the Telephony Services Driver Interface 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

NetWare® OS by the Driver NLM. The Driver should return this buffer back to the TSDI as outlined above.

§7.7. tdiDriverSanity( )

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

Syntax

```
#include <tdi.h>

TDIReturn_t tdiDriverSanity(

        TDIHandle_t   driverID)              /* INPUT  */
```

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 correctly identify which buffer queue to use to access the message from the Tserver.

Return Values

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

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

Driver NLM Notes

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

µ

7.8. tdiQueueSize ( )

This function will return the current status of the message buffers used for the Telephony Services Driver interface. 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

```
#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:

```
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 Telephony Services Driver interface. 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 NLM 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 the Tserver NLM has performed a *tdiAllocBuffer()* or a *tdiReceiveFromDriver()* function call.

Return Values

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

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

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 Telephony Services Driver Interface. 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.

7.9. 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.

Syntax

```
#include <tsrv/tdi.h>

TDIReturn_t tdiMemAllocSize (

    TDIHandle_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. This count includes the (12) bytes that are part of the overhead for each message buffer allocated from the Tserver-Driver interface.

bytes_queued_to_tserver

This parameter specifies the number of bytes in message buffers that are currently queued to the Tserver. This count includes the (12) bytes that are part of the overhead for each message buffer allocated from the Tserver-Driver interface.

bytes_allocd_by_driver

This parameter specifies the number of bytes in message buffers that are currently allocated to the Driver. Message buffers are allocated to the Driver if the Driver NLM has performed a *tdiAllocBuffer()* or a *tdiReceiveFromTserver()* function call. This count includes the (12) bytes that are part of the overhead for each message buffer allocated from the Tserver-Driver interface.

bytes_allocd_by_tserver

This parameter specifies the number of bytes in message buffers that are currently allocated to the Tserver. Message buffers are allocated to the Tserver if the Tserver NLM has performed a *tdiAllocBuffer()* or a *tdiReceiveFromDriver()* function call. This count includes the (12) bytes that are part of the overhead for each message buffer allocated from the Tserver-Driver interface.

Return Values

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

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

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.

µ

7.8. tdiQueueSize ( )

This function will return the current status of the message buffers used for the Telephony Services Driver interface. 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

```
#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:

```
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 Telephony Services Driver interface. 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 NLM 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 the Tserver NLM has performed a *tdiAllocBuffer()* or a *tdiReceiveFromDriver()* function call.

Return Values

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

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

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 Telephony Services Driver Interface. 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.

µ

7.8. tdiQueueSize ( )

This function will return the current status of the message buffers used for the Telephony Services Driver interface. 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

```
#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:

```
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 Telephony Services Driver interface. 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 NLM 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 the Tserver NLM has performed a *tdiAllocBuffer()* or a *tdiReceiveFromDriver()* function call.

Return Values

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

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

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 Telephony Services Driver Interface. 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.

7.9. 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.

Syntax

```
#include <tsrv/tdi.h>

TDIReturn_t tdiMemAllocSize (
```

```
TDIHandle_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. This count includes the (12) bytes that are part of the overhead for each message buffer allocated from the Tserver-Driver interface.

bytes_queued_to_tserver

This parameter specifies the number of bytes in message buffers that are currently queued to the Tserver. This count includes the (12) bytes that are part of the overhead for each message buffer allocated from the Tserver-Driver interface.

bytes_allocd_by_driver

This parameter specifies the number of bytes in message buffers that are currently allocated to the Driver. Message buffers are allocated to the Driver if the Driver NLM has performed a *tdiAllocBuffer()* or a *tdiReceiveFromTserver()* function call. This count includes the (12) bytes that are part of the overhead for each message buffer allocated

from the Tserver-Driver interface.

bytes_allocd_by_tserver

This parameter specifies the number of bytes in message buffers that are currently allocated to the Tserver. Message buffers are allocated to the Tserver if the Tserver NLM has performed a *tdiAllocBuffer()* or a *tdiReceiveFromDriver()* function call. This count includes the (12) bytes that are part of the overhead for each message buffer allocated from the Tserver-Driver interface.

Return Values

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

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

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.

7.10. tdiGetSessionIDInfo()

§This function will return the current information related to an ACS session.

Syntax

```
#include <tsrv/tdi.h>

TDIReturn_t tdiGetSessionIDInfo (
        TDIHandle_t   driverID,        /* INPUT  */
        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 in the Driver space. 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 this session*/
  unsigned long   network;              /* Network of worktop */
  unsigned char   node[6];              /* Node of worktop */
  LoginTime_t    timeOpened;    /* Time the ACS stream was opened */
  char           homeDeviceID[16];   /* Primary device ID of Home
                                        * WorkTop record
                                        */
  char           awayDeviceID[16];   /* Primary device ID of Away Worktop
                                        * record.
                                        */
} TDISessionID_t;
```

Return Values

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

TDI_ERR_BAD_DRVRID	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

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 information about an ACS session that would be usefull in debugging or trace statements from the driver.

7.11. tdiMapInvokeID()

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

Syntax

```
#include <tsrv/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

This is the ACS sessionID assigned to this session by the Tserver 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.

invokeID

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

Return Values

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

TDI_ERR_BAD_DRVRID	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

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 information about an ACS session that would be usefull in debugging or trace statements from the driver.

8. TSDI Header Files tc "Appendix 2: tdi.h"§

8.1. tdi.h

All header files needed for TSDI development are contained on the **CSTA SDK** disk which is part of the **TSDI SDK** kit.

9. ACS and CSTA Message Interface Header Filestc "Appendix 3: csta.h"§

This section describes the C Language header file that defines the ACS and CSTA message interface between the PBX Driver and the application.

9.1. acs.h

All header files needed for TSDI development are contained on the **CSTA SDK** disk which is part of the **TSDI SDK** kit.

9.2. acsdefs.h

All header files needed for TSDI development are contained on the **CSTA SDK** disk which is part of the **TSDI SDK** kit.

9.3. csta.h

All header files needed for TSDI development are contained on the **CSTA SDK** disk which is part of the **TSDI SDK** kit.

9.4. cstadefs.h

All header files needed for TSDI development are contained on the **CSTA SDK** disk which is part of the **TSDI SDK** kit.

10. OA&M API Manual Pagestc "Appendix 4: OA&M API Manual Pages"§

This section 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 Telephony Server. This interface, from the Telephony Servers point of view, 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.

10.1. 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

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

RetCode_t tsrvDriverRequest(



RetCode_t FAR pascal _export
tsrvDriverRequest (    ACSHandle_t                acsHandle,
                                    InvokeID_t                invokeID,

        ACSHandle_t         *acsHandle,         /* RETURN */
        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, i.e. 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 insure 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 Server.

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 Server is not present in the network.

ACSERR_NORESOURCE

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

Comments

None.

Application Notes

None.

10.2. 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..

```
typedef struct
{       ACSHandle_t acsHandle;     EventClass_t  eventClass;     EventType_t  eventType;
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t    eventHeader;
        union
        {
                TSRVDriverConfirmationEvent       driverConfirmation;
        } event;
        char    heap[TSRV_DRIVER_HEAP];
} TSRVDriverEvent_t;

typedef struct
{
        InvokeID_t     invokeID;
        union
        {
                TSRVDriverOAMConfEvent_t        driverConf;
        } u;
} TSRVDriverConfirmationEvent;

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

Parameters

acsHandle

This is the handle for the newly opened ACS Stream.

eventClass

This is a tag with the value **TDRVRCONFIRMATION**, which identifies this message as an Tserver Driver OA&M confirmation event.

eventType

This is a tag with the value **TSRV_DRIVEROAM_CONF**, which identifies this message as an **TSRVDriverOAMConfEvent**.

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.

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.***

Comments

None.

Application Notes

None.

10.3. TSRVDriverOAMEvent

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

Syntax

The following structure describes this event.

```
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];
} 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.***

Comments

None.

Application Notes

None.

11. OA&M Header Filestc "Appendix 5: oam.h"§

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.

11.1. drvrdefs.h

All header files needed for TSDI development are contained on the **CSTA SDK** disk which is part of the **TSDI SDK** kit.

11.2. tdrvr.h

All header files needed for TSDI development are contained on the **CSTA SDK** disk which is part of the **TSDI SDK** kit.

12. Error Log Manual Pagestc "Appendix 6: Error Log Manual Pages"§

The following manual page describes the function call interface for the Error Logger that may be used by the PBX Driver NLMs.

13. Referencestc "References"§

TSAPI *Telephony Server Application Programing Interface (TSAP)* T.A. Anschutz, and J.R. Garcia AT&T Global Business Communications Systems, Issue 1.7 November 29, 1993.

[ECMA/52] *Technical Report ECMA/52, Computer Supported Telecommunications Applications (CSTA),* European Computer Manufacturers Association, June 1990.

[ECMA-179] *STANDARD ECMA-179, Services For Computer Supported Telecommunications Applications (CSTA),* European Computer Manufacturers Association, June 1992.

[ECMA-180] *STANDARD ECMA-180, Protocol For Computer Supported*

Telecommunications Applications (CSTA), European Computer Manufacturers Association, June 1992.

[NLMREF-I] *Novell® NetWare® Loadable Module Library Reference Vol I,* Novell, Inc. September 1991 Edition.

[NLMREF-II] *Novell® NetWare® Loadable Module Library Reference Vol II,* Novell, Inc. September 1991 Edition.

[DRV-SDK] *Telephony Server Driver Software Development Kit*, AT&T Global Business Communications Systems, To Be Completed.

µ

7.3. tdiAllocBuffer()

This function is issued by the Tserver NLM or the Driver NLM to allocate a buffer for sending a message across the Telephony Services Driver Interface.

Syntax

```
#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 12 bytes of this memory block will be used by the TSDI to implement the message queues, however, this 12 bytes* ***should not*** *be included in the* ***length*** *field. The TSDI will automatically add the 12 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:

TDI_EXCEED_HIWATER_MARK The current amount of memory allocated for message buffers (by the TServer NLM and the Driver NLM) is greater than the high water mark (***hiwater_mark***) specified in the *tdiDriverRegister()* function.

TDI_EXCEED_MAX_BYTES	The current amount of memory allocated for message buffers (by the TServer NLM and the Driver NLM) is greater than the maximum number of bytes (***max_bytes)*** specified in the *tdiDriverRegister()* function. This will only be returned on a failure.

Return Values

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

TDI_ERR_BAD_DRVRID	This error indicates that the ***driverID*** specified in the *tdiAllocBuffer()* function is not valid.
TDI_ERR_NO_MEM	This error indicates the TSDI was unable to allocate the requested memory from the NetWare® OS.
TDI_ERR_BADLENGTH	This error indicates that the requested ***length*** is greater than TDI_MAX_BUFFER_SIZE.

TDI_ERR_NO_BUFFERS	This error indicates that the Tserver and the Driver have (together) allocated more memory for message buffers than allowed for this Telephony Services Driver interface. 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.
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

The *tdiAllocBuffer()* function provides a buffer to the Tserver NLM or the Driver NLM that can be used to send a message across the Telephony Services Driver Interface. The buffers are allocated from Netware® OS if the current amount of memory allocated for this interface is less than the maximum specified at Driver registration time. The Driver NLM is responsible for setting the maximum bytes allowed for this interface during the *tdiDriverRegister()* routine. If the Driver NLM 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, or freeing the buffer via the *tdiFreeBuffer()* function. If the Tserver NLM has allocated a message buffer via the *tdiAllocBuffer()* routine, the Tserver is responsible for freeing the buffer back to the Telephony Services Driver Interface either by successfully sending the buffer to the Driver NLM via the *tdiSendToDriver()* 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 NLMs 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 NetWare OS by the Tserver or the Driver NLMs; the messages should be released back to the Telephony Services Driver interface via the *tdiFreeBuffer()* routine.

Warning

The Tserver and Driver NLMs 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 NetWare® OS may not have the resources at this time to fulfill the memory allocation request.

Driver NLM Notes

The Driver NLM 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 NLM 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 NLM is not responsible for the memory resources allocated by *tdiAllocBuffer()* routine since they are allocated from the Novell® NetWare OS in the Tserver context. The Telephony Services Driver interface is responsible for freeing these memory resources. The Driver NLM is responsible for giving the memory resources back to the Telephony Services Driver interface via the *tdiFreeBuffer()* routine, or sending the buffer to the Tserver via the *tdiSendToTserver()* routine.

Tserver NLM Notes

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