PassageWay® Telephony Services
CSTA Services PBX Driver Interface Specification

Lucent Technologies - Formerly the systems and technology units of AT&T
Notice

Every effort was made to ensure that the information in this document was complete and accurate at the
time of printing. However, information is subject to change.

Your Responsibility for Your System's Security

Toll fraud is the unauthorized use of your telecommunications system by an unauthorized party, for
example, persons other than your company’s employees, agents, subcontractors, or persons working on your
company’s behalf. Note that there may be a risk of toll fraud associated with your telecommunications
system and, if toll fraud occurs, it can result in substantial additional charges for your telecommunications
services.

You and your system manager are responsible for the security of your system, such as programming and
configuring your equipment to prevent unauthorized use. The system manager is also responsible for
reading all installation, instruction, and system administration documents provided with this product in
order to fully understand the features that can introduce risk of toll fraud and the steps that can be taken to
reduce that risk. Lucent Technologies does not warrant that this product is immune from or will prevent
unauthorized use of common-carrier telecommunication services or facilities accessed through or connected
to it. Lucent Technologies will not be responsible for any charges that result from such unauthorized use.

Trademarks

Novell, NetWare, and the NetWare logotype are registered trademarks of Novell, Inc.
Microsoft, DOS, Windows, Windows NT and the Microsoft logotype are registered trademarks of
Microsoft Corp.
PassageWay and the Lucent Technologies logotype are registered trademarks of Lucent Technologies.

Disclaimer

Intellectual propert related to this product and registered to AT&T Corporation has been transferred to
Lucent Technologies Incorporated.

Any references within this text to American Telephone and Telegraph Corporation or AT&T should be
interpreted as references to Lucent Technologies. The exception is cross-references to books published
prior to December 31, 1996, which retain their original AT&T titles.

Heritage

Lucent Technologies – formed as a result of AT&T’s planned restructuring – designs, builds, and delivers a
wide range of public and private networks, communication systems and software, consumer and business
telephone systems, and microelectronic components. The world-renowned Bell laboratories is the research
and development arm for the company.

Acknowledgment

This document was prepared by the Business Communications Systems Product Documentation
# Table of Contents

1. Introduction ............................................................................................................................................1  
   1.1. Organization................................................................................................................................2  
   1.2. Overview .....................................................................................................................................2  
   1.3. Terminology ..............................................................................................................................4  
   1.4. Related Documents ...............................................................................................................5  
2. Document Updates .................................................................................................................................6  
   2.1. Change History .........................................................................................................................6  
3. Description .............................................................................................................................................7  
   3.1. Telephony Server Architecture .............................................................................................7  
   3.2. Telephony Server Communication Model .............................................................................8  
   3.3. CSTA Server Tasks and Responsibilities ..............................................................................9  
   3.4. PBX Driver Tasks and Responsibilities ................................................................................9  
   3.4.1. Driver Initialization .............................................................................................9  
   3.4.2. Message-Based Interface .....................................................................................10  
   3.4.3. Driver Termination ..............................................................................................10  
   3.5. CSTA Server Multiplexing ...................................................................................................10  
   3.6. Security .....................................................................................................................................11  
   3.7. Error Log Interface ...............................................................................................................11  
4. Functional Description ...........................................................................................................................11  
   4.1. PBX Driver to CServer Interface ...........................................................................................11  
   4.1.1. Driver Registration ..............................................................................................12  
   4.1.1.1. Registration Mechanism .....................................................................12  
   4.1.1.2. CSDI Memory Allocation ...................................................................13  
   4.1.1.3. Driver Registration Security Level .....................................................13  
   4.1.2. CSDI Version Control .........................................................................................14  
   4.1.3. Receiving Requests and Responses .....................................................................14  
   4.1.4. Sending Requests and Responses ........................................................................14  
   4.1.5. System Status and Heartbeat Message .......................................................................15  
   4.1.6. Unregistering the Driver .......................................................................................17  
   4.1.7. CSTA Services Driver Interface Monitoring .......................................................17  
   4.1.8. Relationship of CSTA Services Driver Interface to TSDI ...................................18  
   4.1.8.1. CSDI and TSDI Message Queues .......................................................18  
   4.1.8.2. PBX Driver Use of TSDI functions .....................................................19
Table of Contents

4.1.9. CSTA Server Flow Control of CSDI Messages ..............................................................19
  4.1.9.1. Telephony Server Flow Control Overview ..........................................................19
  4.1.9.2. Selective Driver Flow Control Via tdISetMessageFlowControl() ......................20

4.2. Messaging Interface ........................................................................................................20
  4.2.1. CSDI Message Format ............................................................................................20
  4.2.2. ROSE Services and Protocol ..................................................................................21
    4.2.2.1. RO-INVOKE Service .........................................................................................21
    4.2.2.2. RO-RESULT Service .......................................................................................22
    4.2.2.3. RO-ERROR Service .........................................................................................22
    4.2.2.4. RO-REJECT-U Service ....................................................................................23
    4.2.2.5. RO-REJECT-P Service ....................................................................................24

4.3. TSDI to ECMA-180 Message Translation .....................................................................24
  4.3.1. CSTA Control Services Functions ..........................................................................24
  4.3.2. CSTA Security Services Functions .......................................................................26
  4.3.3. CSTA Switching Function Services .......................................................................26
  4.3.4. CSTA Status Reporting Services ..........................................................................32
    4.3.4.1. Status Reporting Functions .............................................................................32
    4.3.4.2. Call Event Reports ..........................................................................................33
    4.3.4.3. Feature Event Reports .....................................................................................35
    4.3.4.4. Agent State Event Reports .............................................................................36
  4.3.5. CSTA Snapshot Services .......................................................................................37
  4.3.6. CSTA Computing Function Services .....................................................................37
    4.3.6.1. Routing Registration Functions and Events ......................................................38
    4.3.6.2. Routing Functions and Events .........................................................................38
    4.3.6.3 Register Request ID vs. Routing Cross Reference ID ........................................38
  4.3.7. CSTA Escape/Maintenance Services .....................................................................41
    4.3.7.1. Escape Services: Application as Client ............................................................41
    4.3.7.2. Escape Services: Driver/Switch as Client .........................................................42
    4.3.7.3. Maintenance Services .....................................................................................42
      4.3.7.3.1. Device Status ..............................................................................................42
      4.3.7.3.2. System Status - Application as Client .........................................................43
      4.3.7.3.3. System Status - Driver/Switch as Client ......................................................44
# Table of Contents

4.4. Mapping of CSTA-Defined Types ................................................................. 44
  4.4.1. Device Identifiers ............................................................................... 44
  4.4.2. Call and Connection Identifiers ....................................................... 46
  4.4.3. Connection States ............................................................................ 47
  4.4.4. Status Reporting ............................................................................... 48
  4.4.5. Device and Feature Types and Other Parameters ........................... 51
  4.4.6. Universal failure and Event Cause .................................................. 60
  4.4.7. Security Service ............................................................................... 63
  4.4.8. Common Extensions and Private Data ............................................. 63
  4.4.9. Octet String to Long Identifier Mapping .......................................... 64
  4.4.10. Octet String to Char String Mapping ............................................ 65

4.5. ROSE Error Mappings ............................................................................. 66

4.6. Error Log Interface .................................................................................. 66

5. Compiling and Linking a Driver ................................................................. 67
  5.1. Compiling and Linking For NetWare ..................................................... 67
  5.2. Compiling and Linking For Windows NT ............................................. 67

6. CSDI Coding Examples ............................................................................... 68
  6.1. Registering the Driver with the CSTA Server ....................................... 68
    6.1.1. NetWare Coding Example ............................................................ 68
    6.1.2. Windows NT Coding Example .................................................... 72
  6.2. Receiving a CSTA Request ................................................................. 78
  6.3. Sending a CSTA Response ................................................................. 79

7. CSTA Services Driver Interface Manual Pages ......................................... 81
  7.1. cdiDriverRegister( ) .......................................................................... 81
  7.2. cdiDriverUnregister( ) ....................................................................... 94
  7.3. cdiAllocBuffer() .............................................................................. 95
  7.4. cdiFreeBuffer( ) ............................................................................ 99
  7.5. cdiSendToCSTA( ) .......................................................................... 102
  7.6. cdiReceiveFromCSTA( ) ................................................................. 105
  7.7. cdiDriverSanity( ) .......................................................................... 107
  7.8. cdiQueueSize( ) ............................................................................ 108
  7.9. cdiMemAllocSize() .......................................................................... 110
  7.10. cdiGetCDISize() .......................................................................... 112
  7.11. cdiGetCSTAVersion() ..................................................................... 114
Table of Contents

8. CSDI, ACS, and CSTA Message Interface Header Files ................................................................. 116
   8.1. cdi.h .................................................................................................................................. 116
   8.2. tdi.h ................................................................................................................................ 116
   8.3. acs.h .............................................................................................................................. 116
   8.4. acsdefs.h ....................................................................................................................... 116
   8.5. csta.h ............................................................................................................................ 116
   8.6. cstadefs.h .................................................................................................................... 116
9. References .................................................................................................................................. 116
1. Introduction

This document is the external specification of the CSTA Services Driver Interface for the Telephony Services product. The Telephony Server provides service for Computer-Supported Telecommunications Applications (CSTA) in a networked environment. The Telephony Server consists of three primary parts, the Application Programming Interface (API), the Tserver, and the PBX Driver. An optional fourth part of the Telephony Server, called the Cserver (that is, CSTA Server), provides an alternate PBX Driver interface and simplifies the task of implementing 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 network with the Tserver.

The Tserver is the component that is responsible for routing messages between the application workstations on the network and the PBX Driver. The PBX Driver implements the CSTA services via a switch vendor specific Computer Telephony Integration (CTI) link to a PBX. The Cserver performs certain management and mapping functions on behalf of the PBX Driver.

The CSTA Services Driver Interface (CSDI) is an open interface that specifies how messages are passed between the Cserver and a vendor's PBX Driver, and is based on the ECMA-180 CSTA standard. The Telephony Services Driver Interface (TSDI) is also an open interface and specifies how messages are passed between the Tserver and the Cserver (or between the Tserver and a PBX Driver when the Cserver is not used). The TSDI is described in detail in [TSDI]. A PBX Driver vendor may choose to implement either interface. The CSDI interface is described in this document.

The Cserver's main function is to free the PBX Driver from the task of managing client sessions. A PBX Driver using the CSDI is aware of only one computing domain application for which it provides services. This is the model provided by the CSTA standard. The Cserver also provides a true CSTA (that is, ECMA-180 compliant) interface. That is, APDUs (application protocol data units) passed across the CSDI are encoded using the transfer syntax defined by ASN.1 Basic Encoding Rules (BER) and conform to the CSTA and ROSE (Remote Operations Service Element) protocols. In contrast, the TSDI interface specifies a C Language structure message format.

This document is intended for use by PBX vendors that wish to write a PBX Driver that conforms to the CSDI interface. It specifically covers the function call interface to pass the messages between the components, the structure of the messages, and requirements that vendors must follow when writing a PBX Driver that adheres to the interface. It is assumed throughout this document that the reader is familiar with the Telephony Services Driver Interface (TSDI).

Release 2 of Telephony Services is available on both the Novell® NetWare® platform and the Microsoft® Windows NT® platform. This document describes the common CSTA Services Driver Interface for both platforms and any differences between platforms are noted.

This document refers to the Tserver, PBX drivers, and the Cserver as “components.” Note that the general architecture of Telephony Services on NetWare dictates that components such as the Tserver and drivers are NLMs (NetWare Loadable Modules), whereas on Windows NT the Tserver is a service and drivers are DLLs (Dynamic Link Libraries). In addition, “service advertising” in this document refers both to the method used over the Novell network to advertise a service, as well as to the method on the TCP network of querying a known name server for available services.
1.1. Organization

Section 1 describes how the CSTA-Server 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 2 describes changes that have been made to this document since the initial version.

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 Cserver for the PBX Driver.

Section 4 gives a detailed description of the CSTA Services Driver Interface from the PBX Driver perspective.

Section 5 discusses compiling and linking the PBX Driver.

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

Section 7 contains manual pages for the CSTA Services Driver Interface functions.

Section 8 describes the CSDI, ACS, and CSTA message interface header files.

Section 9 lists all of the documents referenced in this specification.

1.2. Overview

The Telephony Server provides the desktop integration of telephones and personal computers by exporting a Computer-Supported Telecommunications Applications (CSTA) API over a network. Figure 1 provides a high level view of a simple Telephony Server configuration. Each phone and associated computer represents a workstation on a user'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 network with the Tserver.

The Tserver is a software module that resides on a network Server. The Tserver is responsible for routing messages between the applications on workstations connected to the network and the PBX Driver. In some instances, the Tserver may not interface directly with the PBX Driver. Instead, it interfaces with a module called the Cserver, which in turn interfaces with the PBX Driver.

The Cserver is a software module that resides on the network Server and performs certain functions on behalf of the PBX Driver. The Cserver manages client sessions and multiplexes multiple client service request streams appearing at the TSDI into a single stream across the CSDI. The Cserver presents a true CSTA interface (that is, ECMA-180 compliant) to the PBX Driver. Messages exchanged across the CSDI conform to the CSTA and ROSE (Remote Operations Service Element) services and protocol as specified in standards ECMA-179, ECMA-180 and CCITT Recommendations X.219 and X.229. ROSE provides for a simple request/response protocol and defines a mechanism for associating responses with prior requests.

The PBX Driver is a software module that resides on the network Server and implements the CSTA services via a switch-vendor-specific Computer Telephony Integration (CTI) link to a PBX. A CTI link is a logical link between the computing environment (Telephony Server) and the switching environment (PBX). Each driver that supports CTI links via the CSDI or TSDI must define the physical implementation for a CTI link (that is, the CTI link could be one or more physical links). 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.

The PBX Driver is a software module that resides on the network Server and implements the CSTA services via a switch-vendor-specific Computer Telephony Integration (CTI) link to a PBX. A CTI link is a logical link between the computing environment (Telephony Server) and the switching environment (PBX). Each driver that supports CTI links via the CSDI or TSDI must define the physical implementation for a CTI link (that is, the CTI link could be one or more physical links). 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.

The PBX Driver is a software module that resides on the network Server and implements the CSTA services via a switch-vendor-specific Computer Telephony Integration (CTI) link to a PBX. A CTI link is a logical link between the computing environment (Telephony Server) and the switching environment (PBX). Each driver that supports CTI links via the CSDI or TSDI must define the physical implementation for a CTI link (that is, the CTI link could be one or more physical links). 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.

The CSTA-Server Driver Interface (CSDI) is an open interface that specifies how messages are passed between the Cserver and the PBX Driver. The PBX driver should only use one of the TSDI or CSDI interfaces for CSTA services.
For OA&M Services, the PBX Driver should always use the Telephony Services Driver Interface (TSDI). There is no support for OA&M Services in the CSTA Services Driver Interface.

![Telephony Server Connectivity Diagram]

Figure 1: Telephony Server Connectivity
1.3. Terminology

APDU

*Application Protocol Data Unit.* A data object exchanged between the CSTA Server and PBX Driver, or between the Telephony Server and the client application.

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.

ASN.1

*Abstract Syntax Notation One.* A notation for the definition of abstract syntaxes defined in CCITT Recommendation X.208., enabling application layer specifications to define the types of information they need to transfer. ASN.1 also specifies a notation for the specification of values of a defined type.

BER

*Basic Encoding Rules.* A set of encoding rules defined in CCITT Recommendation X.209 that may be applied to values of types defined using the ASN.1 notation. Application of these encoding rules produces a transfer syntax for such values.

Cserver

*CSTA Server.* The specific module that manages the routing of CSTA requests and responses between the Tserver and a PBX driver. The Cserver is part of the Telephony Server and performs certain management and mapping functions on behalf of the PBX Driver.

CSDI

*CSTA Services Driver Interface.* A connection between a PBX Driver and the Cserver using a function call interface to pass ASN.1/BER encoded CSTA APDUs between modules.

CTI

*Computer Telephony Integration*

DLL

*Dynamic Link Library.* A module that can be dynamically loaded on the Windows NT Operating System.

NLM

*NetWare Loadable Module.* 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.

ROSE

Remote Operations Service Element. The application service element defined in CCITT Rec. X.219. Defines a model, notation, services, and protocol for the exchange of requests/responses carried out within the context of an application-association.

Telephony Server

The Telephony Server supports CSTA in a networked environment. The Telephony Server may also refer to a network server that has been loaded with and runs the Telephony Services software.

Telephony Services

Telephony Services refers to the entire product, including all software modules on the Server and the various client libraries. Telephony Services is available for both Windows NT and Novell NetWare.

TSDI

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

Tserver

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

1.4. Related Documents

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

[TSAPI]  
Telephony Services Application Programming Interface (TSAPI) Definition and Specification.

[TSDI]  
Telephony Services: PBX Driver Interface Specification Releases 1.0 and 2.0

[ECMA/52]  

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

[ECMA-180]  

[X.208]  
Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1). The International Telegraph and Telephone Consultative Committee (CCITT)
2. Document Updates

2.1. Change History

Issue 1.0 was the first version of the CSTA Services Driver Interface Specification.

Issue 1.1 contains the following changes to the CSDI:

- Enhancements to support a low-water mark on buffer usage.
- Requirement added on PBX Drivers to notify the Cserver of terminated monitors.
- Enhancements to allow support of future versions of the CSTA Protocol.
- Changes to the mappings of the CSTAPrivateData and CSTACommonArguments types.
- Addition of previously omitted error return codes for the cdiDriverRegister() and cdiReceiveFromCSTA() routines.

Note that as a result of the above changes the cdiDriverRegister() and cdiGetCSTAVersion() interfaces have been changed.

Issue 1.2 reflects solely the production changes (cover and footer information) required to prepare this document for the November 1994 "Alpha Delivery" to Beta ISVs.

Issue 1.3 contains an updated title page and cover page for the February 9, 1995 delivery to Novell.

Issue 1.4 contains the following changes to the CSDI:

- Enhancements for TSAPI version control and private data version negotiation.
- CSDI settings (that is, max_bytes, hiwater-mark, and lowater-mark) are now saved in the Tserver database.

Issue 1.5 contains editorial changes.

Issue 1.6 reflects the changes for Telephony Services on the Windows NT platform.

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 users’ phones. The CSTA
applications will use a library (the CSTA API) as a client communicating with a Telephony Server running on either a Novell NetWare or Windows NT 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 also provides a separate set of generic OA&M services to OA&M clients.

![Diagram of Telephony Server](image)

**Figure 1: Block Diagram of Telephony Server**

The Telephony Server runs under both Novell NetWare and Windows NT (the large box in Figure 1). The three main Telephony Server modules that run on the server 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 message) and authenticate some CSTA (but not OA&M) requests. In addition, it will route the requests either to the Cserver or to the appropriate PBX Driver.
Cserver

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

PBX Driver

The PBX driver handles CSTA and/or OA&M requests for a specific vendor’s PBX. The PBX Driver may choose to provide CSTA services through the Tserver using the TSDI interface, or through the Cserver, using the CSDI interface. The PBX Driver may choose to provide OA&M services through the Telephony Server, or the driver may choose to provide its own OA&M interface.

The interface between the Cserver and the PBX Driver is a function call interface referred to in this document as the CSTA Services Driver Interface (or CSDI). The CSTA Services Driver Interface is used to pass APDUs that represent CSTA requests and responses between the CSTA Server (Cserver) and the PBX Driver. The format of APDUs exchanged over the CSDI interface is defined by the following international standards:

- Standard ECMA-180: Protocol for Computer-Supported Telecommunications Applications (CSTA)
- Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)

The messages are pure ROSE (Rec. X.229) APDUs not encapsulated by any other protocols such as ACSE (Rec. X.227) or RTSE (Rec. X.228). The following three ROSE operation classes will be utilized, as specified in ECMA-180:

- Operation Class 2 - Asynchronous, reporting success or failure (result or error).
- Operation Class 3 - Asynchronous, reporting failure (error) only, if any.
- Operation Class 5 - Asynchronous, outcome not reported.

The Telephony Server provides services to clients distributed over the network. The Tserver handles service advertising, client authentication, connection setup and connection tear down. All CSTA client requests are multiplexed to a single stream per PBX driver registration; that is, each cdDriverRegister call at the CSDI interface is translated into a tdiDriverRegister call at the TSDI interface and generates a different driver ID which the driver then uses in cdReceiveFromCSTA calls. Any CSTA client requests which come over ACS streams that were opened up to the advertised name created on behalf of this registration are multiplexed to the single stream of messages that a driver receives by using cdReceiveFromCSTA. Client ACS requests are handled entirely within the Cserver and are not seen by the PBX driver. A PBX driver may register for CSTA services one or more times. Each 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.
3.3. CSTA Server Tasks and Responsibilities

The two main responsibilities of the CSTA Server are client session management and the mapping of messages between the CSDI and TSDI formats. Client session management encompasses supporting all of the ACS services (that is, `acsOpenStream()`, `acsCloseStream()`, etc.) defined by TSAPI within the CSTA server in order to free the PBX Driver from having to track individual clients. The Cserver must keep track of all client sessions and associated service requests (including the `invokeID` exchanged across the CSDI) so that responses to requests made by a particular client can be sent back to the same requesting client. The Cserver must also maintain a mapping between all CSTA monitors that have been established and the associated client application sessions (that is, between the `monitorCrossReferenceID` assigned by the switch/driver and the `sessionId`) so that CSTA events can be routed to the appropriate client application (that is, the application session that established the monitor for which the event was generated). The message mapping function encompasses mapping of the C-structure format messages exchanged across the TSDI into ECMA-180 APDUs excanged across the CSDI, and vice-versa. The details of these mappings are specified in section {4.2.1. CSDI Message Format} of this document.

Besides the two main responsibilities discussed above, the CSTA Server is also responsible for supporting the TSAPI routing registration and system status services which have no direct counterpart in ECMA-179. The routing registration services allow clients to register to become the routing server for a particular routing device, or to become the default routing server. The system status services allow clients to query for system status, report application system status, and register for system status events. These two classes of services must be supported by the Cserver on behalf of a PBX driver that registers with the Cserver.

3.4. PBX Driver Tasks and Responsibilities

The design of a PBX Driver for the Telephony Server begins when the vendor determines the CSTA services the driver will support and which interface the driver will use. 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 reject any CSTA client request that it does not support.

The following sections describe three tasks that the PBX Driver must implement to support CSTA services. It is assumed from this point on that the PBX Driver is using the CSDI interface.

3.4.1. Driver Initialization

The PBX Driver must register with the Cserver before CSTA requests can be routed to the driver. When the PBX Driver registers with the Cserver, resources are allocated for the CSTA Services Driver Interface, and the Cserver registers the driver with the Tserver to establish a Telephony Services Driver Interface (TSDI) between the Cserver and Tserver on behalf of the registering PBX driver.

The Tserver will advertise on the network that the driver is available to handle CSTA (CTI Link) application requests. The PBX Driver must separately register with the Cserver for each CTI Link the driver is going to support. If the PBX Driver supports OA&M services, it must register directly with the Tserver for these services. It is perfectly legal for the PBX Driver to register through the Cserver for CSTA services and through the Tserver for OA&M services simultaneously. Each registration results in a completely separate interface being created, and message traffic across distinct interfaces are completely independent.

The Telephony Server supports both the CSDI and TSDI simultaneously. It is permissible to load multiple PBX Drivers in a single Telephony Server with some registering through the CSDI and others through the TSDI. A PBX Driver must not register with both the CSDI and TSDI for the same CTI Link, however. The Cserver always registers with the TSDI on behalf of the PBX Driver. The Cserver will support the same number of driver registrations that the Tserver supports.

There are slightly different mechanisms for starting (loading) a PBX driver on a Windows NT server and on a Novell NetWare server. These different mechanisms are fully described in [TSDI]. Although the mechanisms differ slightly between platforms, they are independent of whether the driver registers for service through the TSDI or registers for CSTA services through the CSDI.
3.4.2. Message-Based Interface

The PBX Driver must be able to handle incoming requests from the Cserver immediately after the driver has completed the registration process. The PBX Driver sends and receives APDUs that represent ECMA-179 requests and responses to the Cserver through a function call interface. The PBX Driver must always be prepared to reject application requests that the driver does not understand.

NOTE: The messages exchanged over the CSTA Services Driver Interface (CSDI) must be ECMA-180 compliant messages, encapsulated by ROSE (X.229) protocol, and encoded according to Basic Encoding Rules for ASN.1 (X.209). Requests made by CSTA client applications are translated to ROSE RO-INVOKE service requests across the CSDI. Typically, the PBX Driver responds to the request by invoking the ROSE RO-RESULT service. The PBX Driver may also respond by invoking either the RO-ERROR service or the RO-REJECT-U service. The RO-REJECT-U service is invoked by the PBX Driver in response to an unrecognized or unsupported service request. The RO-ERROR service is used in the event of a CSTA error. The Cserver may invoke the RO-REJECT-U service for unrecognized service requests or responses generated by the PBX Driver.

3.4.3. Driver Termination

The PBX Driver should always unregister with the Cserver before exiting (unloading), or the driver can unregister with the Cserver any time it wants to stop handling CSTA requests. The Cserver will unregister the driver with the Tserver, which in turn will halt the advertisement of the driver services, and free all resources associated with the Telephony Services Driver Interface, when the PBX Driver unregisters. Additionally, the Cserver will free all resources associated with the CSTA Services Driver Interface. The PBX Driver cannot use the CSTA Services Driver Interface for the CTI Link for which the register was originally done after the unregister operation has completed. The driver can, however, re-register for CSTA services.

There are slightly different mechanisms for unloading a PBX driver on a Windows NT server and on a Novell NetWare server. These different mechanisms are fully described in [TSDI]. Although the mechanisms differ slightly between platforms, they are independent of whether the driver registers for service through the TSDI or registers for CSTA services through the CSDI. It is important that the PBX driver terminate all threads and free all allocated resources before exiting.

3.5. CSTA Server Multiplexing

The CSTA Server (Cserver) provides a limited form of multiplexing services for the driver. The Cserver accepts and replies to ACS open, close, and abort stream requests made by client applications without involvement of the driver. The PBX Driver is not aware of individual client applications or streams. All service requests made over the collection of client streams managed by the Cserver are multiplexed to a single request stream over the CSDI. The Cserver also frees the PBX driver from client TSAPI protocol version management by implementing TSAPI version control. Messages exchanged over the CSDI conform to a single CSTA protocol version specified by the driver.

Multiple CSTA monitor start service requests associated with the same monitored object are not multiplexed by the Cserver into a single monitor request to the driver. This is true regardless of whether the monitor request was made over the same client stream or not. The driver must be prepared to receive multiple monitor requests for the same monitored object, and must send multiple event reports associated with the monitored object if it chooses to accept the multiple monitor start requests.

Multiplexing is provided for the system status reporting service. Up to 50 client applications can register to receive system status event notifications. These registration requests are handled entirely by the Cserver. When the driver informs the Cserver of a system status change via the CSTA system status service, the Cserver will inform all registered clients of the new status.

A form of multiplexing is also provided for application routing services. Client applications can register to become the routing server for a particular routing device, or to become the default routing server. The registration requests are
handled entirely by the Cserver without the involvement of the driver. When the driver initiates a routing dialogue the route request message(s) are routed to the client application (that is, acting as the server) that has registered to receive route requests for the particular routing device within the request. In the case where no client application has registered for a particular routing device but a client application has registered to be the default routing server, requests are sent to the default routing application. There can be at most one default routing server for a single CSDI interface instance.

3.6. Security

The Cserver does not provide any security services of its own. However, since the Cserver registers the driver with the Tserver, and all requests handled by the Cserver must pass through the Tserver, the Tserver security services described in [TSDI] apply. 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 cdiDriverRegister() routine. A user wishing to open an ACS stream will always require at least a valid Login on the server the Tserver is loaded on. The lowest level of security a driver can register with is CDI_NO_SECURITY which means the Tserver will not validate any message including an acsOpenStream() call; that is, a client may open an ACS stream without a valid login/password. NOTE: For Telephony Services on the NetWare platform, Novell has stated that a driver which uses this level of security will not receive Novell NLM Certification. This level may be useful, however, when developing the driver. The next level of security a driver can request during the registration is CDI_LOGIN_SECURITY which requires the same Login and password on the server and an entry in the TServer Security Data Base. The highest level of security, CDI_CSTA_SECURITY, encompasses the requirements of the previous level and in addition validates CSTA requests against the Security Data Base administration. See the Telephony Services Driver Interface [TSDI] specification for more details on the Telephony Services security service.

The ECMA-179-defined security services are not supported by the Cserver. The Cserver never encodes any CSTA security data in the APDUs sent to the PBX driver and ignores any CSTA security data that may be present in any APDUs received from the PBX driver.

3.7. Error Log Interface

A common error log will serve the Cserver, Tserver, and the PBX driver. The interface to the error log is always through the Telephony Services Administrator application. The application 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, and FATAL (see [TSDI]) for errors, and will include the date, time, location of the error, a specific error code and supporting text for each error. For Telephony Services on the NetWare platform, there are three possible destinations for each severity level: the NetWare system console, the Telephony Services Administrator application, and the error log file. For Telephony Services on the Windows NT platform the possible destinations are: the Windows NT System Event Log, the Telephony Services Administrator application and the error log file. These destinations can be set for each of the six severity levels through the Telephony Services Administrator. Whenever possible, the PBX driver should log errors in the common error log with the appropriate severity. The PBX Driver should call the TSDI error logging routine directly (see section 4.6). There is no error logging routine defined in the CSTA Services Driver Interface (CSDI). PBX Driver authors are free to ignore the TSDI 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 Cserver Interface

The CSDI provides a function call interface that offers a mechanism for the exchange of CSTA protocol messages representing requests and responses that map to the CSTA API (see [TSAPI]). The PBX Driver must first register with the Cserver, and then the driver can send and receive these requests and responses using CSTA Services Driver Interface routines. All messages exchanged with the Cserver must be allocated via the cdiAllocBuffer() routine (See section 7.3) from the CSDI; they cannot be directly allocated from the Operating System or through the TSDI buffer allocation routine (that is, tdiAllocBuffer()). The CSDI and TSDI each manage a separate pool of buffers that
are distinct (that is, a TSDI buffer cannot be passed across the CSDI and a CSDI buffer cannot be passed across the
TSDI). The PBX Driver should always unregister with the Cserver before it is unloaded.

The following sections provide a brief description of the function call interface provided by the CSTA 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 Cserver before it can provide the CSTA services described in the
CSTA API TSAPI. If a driver wants services advertised for a link it supports, that link must be registered separately with
the Cserver; that is, each CSDI registration supports one CTI link.

NOTE: A CTI Link is a logical link connecting the driver to the PBX. The CTI Link can be one or more physical
links. A driver should register separately for each CTI Link that it will support.

The driver must use the CSTA Services Driver Interface routine, cdiDriverRegister(), to establish a connection
between the driver and the Cserver. When a PBX Driver registers via the cdiDriverRegister() routine, the
Cserver creates a separate CSDI instance by allocating the resources from the Operating System for this CSTA Services
Driver Interface (CSDI). The Cserver registers with the Tserver on behalf of the PBX Driver, creating a Telephony
Services Driver Interface (TSDI) instance. In response, the Tserver begins Service Advertising on behalf of the PBX
Driver. The Telephony Server treats each cdiDriverRegister(), or CSDI instance completely independently.
From the Telephony Server’s point of view, each cdiDriverRegister() represents a different driver, even though
multiple registrations may have been done by the same driver.

The PBX Driver must supply a vendor_name and service_name when registering with the Cserver using the
cdiDriverRegister() function. These parameters are passed directly to the Tserver when the Cserver registers
with the Tserver on behalf of the driver. The Cserver always registers with the Tserver with service_type
TDI_ST_CSTA, since a CSDI interface is only capable of supporting CSTA services. The Tserver will apply the
vendor_name, service_name and service_type parameters to the appropriate routine(s) to advertise to clients that the
Driver is available to perform services. The Tserver will also store the driver_name parameter provided by the PBX
Driver in the cdiDriverRegister() routine for maintenance purposes.

Version information must be specified in the cdiDriverRegister() routine so that the PBX Driver can guard
against version compatibility problems with the Cserver. Because only one version of the CSDI currently exists, the
csdi_version field must always be set to CDI_CSDI_VERSION_1. This is also true for the CSTA protocol_version
field, which must always be set to CDI_PROTOCOL_VERSION_1. These two fields together will allow for support
of future CSDI and CSTA protocol versions.

The CSTA Services Driver Interface monitors CSDI buffer usage by both the PBX Driver and the Cserver. The PBX
Driver can specify a maximum number of bytes that can be allocated (by the PBX Driver and the Cserver) for CSDI
messages via the buffer_descriptor parameter of the cdiDriverRegister() routine. This buffering mechanism is
identical to that provided by the Telephony Services Driver Interface (TSDI), with the exception that the CSDI provides
for a low-water mark on buffer usage. See section {4.1.1.2} below for a detailed description of how memory is used by
the CSDI.

Information regarding the mapping of octet strings representing call identifiers, monitor cross reference identifiers, and
routting cross reference identifiers passed across the CSDI must be supplied at registration time. This information is used
by the Cserver to map between the different representations and encodings for these identifiers within ECMA-180 and
TSAPI. See section {4.4.9.} below for a detailed description of how this information is used.

If the PBX Driver supports CSTA Private Data, CSTA Escape Services, or CSTA Private Events, it must provide a
private data vendor identifier, and optionally a list of private data versions numbers, at registration time. The identifier is
a string of up to 32 characters in length and is passed in the priv_data_vendor parameter to cdiDriverRegister().
A list of private data version numbers can be appended at the end of the priv_data_vendor string. Whenever a client
application makes a request that includes private data, the vendor field of the PrivateData_t structure is compared with the identifier supplied in the priv_data_vendor parameter. If they match, the private data is encoded in the APDU sent to the driver; otherwise it is discarded. A reverse mapping occurs for APDUs sent by the driver to the Cserver. In this case, the priv_data_vendor parameter value is used to set the vendor field of the PrivateData_t structure. Only one priv_data_vendor identifier may be associated with a CSDI interface instance. The optional list of private data version numbers is used by the Cserver to implement TSAPI private data version negotiation during processing of a client open stream request. For more information on TSAPI private data version negotiation, refer to TSAPI.

The PBX Driver is also required to pass a pointer to a structure of type CDIDriverCaps_t in the driver_caps parameter. The driver indicates which CSTA services it supports using this parameter. The Cserver responds to a client application’s cstaGetAPICaps() request using this information.

The cdiDriverRegister() routine returns a driverID to the PBX Driver that must be used to identify this CSDI connection. The value returned for driverID is the same as that returned by the Tserver tdiDriverRegister() routine. All message buffer allocations, send requests, receive requests, and unregister requests for this CSDI connection (registration) must use this driverID. The PBX Driver is not allowed to interchange messages from one CSDI registration to another. The cdiDriverRegister() routine is a blocking function that will return to the PBX Driver after the Tserver has initiated the Service Advertising procedures and all resources have been allocated for the TSDI and CSDI interfaces.

4.1.1.2. CSDI Memory Allocation

The Cserver will allocate space from the CSDI interface only for buffers that are used to forward/send events to the driver. All other space allocated by the Cserver is not charged against the total CSDI space. Note that a successful registration with the Cserver means a successful registration has occurred with the Tserver, and a TSDI interface has been allocated. The Cserver will use the same buffer_descriptor parameter values for allocating TSDI space as the driver requested for the CSDI interface. This means that the total allocation of buffer space within the Telephony Server for this driver registration will actually be twice what is indicated in the buffer_descriptor parameter. The CSDI and TSDI interfaces are completely independent and do not share common buffers.

Both the CSDI and TSDI memory allocation parameter values can be changed "on the fly" via the OA&M interfaces provided by the Cserver and Tserver, respectively (that is, using the TSA application). Furthermore, the values of the buffer_descriptor parameter supplied by the PBX driver should be viewed as the initial values only. When a driver registers for the first time, the buffer_descriptor parameter values are maintained by the Tserver in permanent storage (a database on NetWare, the registry on Windows NT). On subsequent registrations, the values stored override the values supplied by the driver in the cdiDriverRegister() function call. This operation allows for tuning of the memory allocation parameters without the need to change the PBX Driver software and allows for the changes to "stick" across loads and unloads of the PBX Driver.

4.1.1.3. Driver Registration Security Level

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

<table>
<thead>
<tr>
<th>CDI_CSTA_SECURITY</th>
<th>Network server Login and Password are validated on the acsOpenStream() request.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Entry in the Tserver's Security Database must contain this login. This is also checked at the time of the acsOpenStream() request.</td>
</tr>
<tr>
<td></td>
<td>Each subsequent CSTA request are validated per the user’s administered permissions.</td>
</tr>
</tbody>
</table>
CDI_LOGIN_SECURITY  Network server Login and Password are 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.

CDI_NO_SECURITY  No validation is done on an `acsOpenStream()`
request.

4.1.2. CSDI Version Control

The `csdi_version` field in the `cdiDriverRegister()` function is used to enforce version control of the CSDI. Currently only one version, CDI_CSDI_VERSION_1, of the CSDI exists. The `csdi_version` field must be set to CDI_CSDI_VERSION_1.

The `protocol_version` field in the `cdiDriverRegister()` function is used to select the CSTA protocol version to be used for the duration of the driver registration. Currently only one version of the CSTA protocol is supported. The `protocol_version` field must be set to CDI_PROTOCOL_VERSION_1. This value indicates the June 1992 version of the CSTA protocol.

The function `cdiGetCSTAVersion()` allows a driver to determine the customer and internal version of the CSTA Server, as well as the list of supported CSDI and CSTA protocol versions. This allows a driver to take advantage of any new functionality in a new release of the Cserver.

4.1.3. Receiving Requests and Responses

The `cdiReceiveFromCSTA()` routine is used by the PBX Driver to receive incoming requests and responses from and to Client applications. These client requests and responses (APDUs) are encoded according to `ASN.1 Basic Encoding Rules` in message buffers (see section {4.2.3} below) returned by the `bufptr` parameter to the `cdiReceiveFromCSTA()` routine. The `cdiReceiveFromCSTA()` routine is a blocking routine that will only return when a message buffer is ready for the PBX Driver or if an error has occurred. The PBX Driver "owns" the message buffer returned by the `cdiReceiveFromCSTA()` routine. The message buffer should not be directly returned to the Operating System by the PBX Driver (e.g. via the `free()` routine). The message buffer must be returned to the CSTA Services Driver Interface. Recall that if a driver wishes to, it is allowed to register with the Cserver multiple times.

The `cdiFreeBuffer()` routine can be used to return the message buffer back to the CSDI, or the message buffer may be populated with a request or response APDU and sent back across the CSTA Services Driver Interface to the client (see section {4.1.3. Receiving Requests and Responses}). The `cdiFreeBuffer()` 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 `cdiReceiveFromCSTA()` routine must be applied to the `cdiFreeBuffer()` routine.

4.1.4. Sending Requests and Responses

The `cdiSendToCSTA()` routine is used by the PBX Driver to send ECMA-180 APDUs to the Cserver. These APDUs are encoded using the transfer syntax defined by `ASN.1 Basic Encoding Rules` in message buffers (see section {4.2.1. CSDI Message Format} below) pointed to by the `bufptr` parameter of the `cdiSendToCSTA()` routine. The message buffers must be "owned" by the PBX Driver, and they must be allocated from the CSTA Services Driver Interface that will be used to send the messages to the Client. (The `driverID` returned from the `cdiDriverRegister()` 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 Cserver (via the `cdiReceiveFromCSTA()` routine as described above in section {4.1.3.}, or allocated the message buffer from the CSDI via the `cdiAllocBuffer()` routine.
The `cdiAllocBuffer()` 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 `cdiAllocBuffer()` routine will fail the “request” if the size of the message buffer requested exceeds the maximum buffer size allowed on the interface (in NetWare 3.x systems, the limit is 4K; there is no limit in NetWare 4.x systems or for Windows NT systems), or if the size requested plus the size of all message buffers currently allocated by the PBX Driver and the Cserver (on this CSTA Services Driver Interface) exceeds the limit specified by the driver during PBX Driver registration. The CSDI memory allocation for this driver (or CSDI registration) will be charged the size of the data block allocated.

The `cdiSendToCSTA()` routine supports a two-level message priority scheme. The PBX Driver can send “priority” messages through the interface by setting the `priority` parameter to “CDI_PRIORITY_MESSAGE”. The CSTA 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 Cserver via `cdiSendToCSTA()`, and the PBX Driver should no longer access this buffer. The `cdiSendToCSTA()` routine is a non-blocking routine that will fail only when the `bufptr`, `priority`, or `driverID` parameters are invalid.

NOTE: The `cdiReceiveFromCSTA()` routine is blocking because it waits on a Semaphore that is only signaled when the Cserver sends the driver a message. The `cdiSendToCSTA()` routine, however, is non-blocking because it signals a different Semaphore on which the Cserver waits to receive messages from the driver.

### 4.1.5. System Status and Heartbeat Message

The PBX Driver must inform the Cserver, once a minute, that it is still active either by sending a system status message (that is, ECMA-180 System Status Request APDU) or by calling the `cdiDriverSanity()` function. The `cdiDriverSanity()` function requires one parameter, the `driverID`, returned to the PBX Driver by the `cdiDriverRegister()` routine. This is a non-blocking routine. If the driver fails to send system status or call the `cdiDriverSanity()` function once a minute, the Tserver will generate a high severity error message which by default is placed in the error log file and sent to the Tserver Administrator application. In the current version of the CSDI, no other recovery action is taken. The PBX Driver should not call the Tserver `tdiDriverSanity()` routine.

System status messages also serve to report the state of the switching system to the computing system, and vice-versa. The following table shows the possible system status values defined in ECMA-179 that may be sent by the PBX Driver to the Cserver. The meaning of each value (taken directly from the ECMA-179 standard text) is indicated in the 2nd column of the table, and the action taken by the Cserver upon receipt of the system status message is indicated in the 3rd column. The action is only taken when the system status changes, but multiple successive reports containing the same state value are permitted. The initial state immediately after driver registration is assumed to be `initializing`.

```markdown
<table>
<thead>
<tr>
<th>Status Value</th>
<th>Description</th>
<th>Action Taken by Cserver</th>
</tr>
</thead>
<tbody>
<tr>
<td>initializing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>System Status Value</th>
<th>Meaning</th>
<th>Cserver Action</th>
</tr>
</thead>
</table>
| disabled            | Existing Monitor Requests have been disabled. Other Requests and Responses may also be disabled, but reject responses should be provided. | 1. Send CSTAMonitorEndedEvents for any existing monitors.  
2. Send CSTARouteRegisterAbortEvents for any existing route registrations.  
3. Stop accepting new route registration requests with error GENERIC_SYSTEM_AVAILABILITY.  
4. Stop accepting new Open Stream requests with error DRIVER_LINK_UNAVAILABLE. |
| enabled             | Requests and responses are re-enabled, usually after a disruption or restart. This status indication shall be sent after an Initializing status indicator has been sent and may be sent under other conditions. This status indicates that there are no outstanding monitor requests. | 1. Send CSTAMonitorEndedEvents for any existing monitors.  
2. Send CSTARouteRegisterAbortEvents for any existing route registrations.  
3. Begin accepting new route registration requests.  
| initializing        | The system is re-initializing or restarting. This status indicates that a system is temporarily unable to respond to any requests. If provided, this status message shall be followed by an Enable status message to indicate that the Init process has completed. | 1. Send CSTAMonitorEndedEvents for any existing monitors.  
2. Send CSTARouteRegisterAbortEvents for any existing route registrations.  
3. Stop accepting new route registration requests with error GENERIC_SYSTEM_AVAILABILITY.  
4. Stop accepting new Open Stream requests with error DRIVER_LINK_UNAVAILABLE. |
| messages lost       | Requests and/or responses may have been lost, including Event Reports. | None. (Note: Error log entry is made.)                                                             |
| normal              | Sent at any time to indicate that the status is normal. This status has no effect on other Services. | None.                                                                                             |
| overload imminent   | Receiver is requested to take initiative to shed load. | None. (PBX Driver may reject new requests if it wishes).                                           |
| overload reached    | Requester may take initiative to shed load. This cause may be followed by Stop Monitor requests sent to the client, and rejections to additional Service Requests. | None. (PBX Driver may reject new requests if it wishes.)                                           |
| overload relieved   | Overload condition has passed. | None.                                                                                             |

The Cserver positively acknowledges all System Status requests issued by the PBX Driver that contain a valid System Status value (that is, cause). In all other cases a CSTA error is returned containing the error qualifier operationalErrors: valueOutOfRange. In addition to the above actions, a change in system status is reported to any application that has registered for system status notifications via the CSTASysStatStart service request.
The PBX Driver must continue to receive and either process or reject client service requests even after reporting a non-normal status to the Cserver. This eliminates the possibility of any race conditions existing between the driver and Cserver in handling requests following such a report, and allows the driver to specify the exact cause of a service request rejection.

System status requests can also be sent by the Cserver to the PBX Driver. The following table shows the system status values, their meaning, and action that should be taken by the PBX Driver in a fashion similar to the table above.

<table>
<thead>
<tr>
<th>System Status Value</th>
<th>Meaning</th>
<th>PBX Driver Action</th>
</tr>
</thead>
</table>
| enabled             | Request for PBX Driver to clear all active monitors and routing dialogs. | 1. Clear all active monitors.  
2. Clear/end all active routing dialogs. |
| normal              | Used as a loop back or “heartbeat” message.  | None.                                    |

The PBX Driver should positively acknowledge the System Status request sent by the Cserver.

### 4.1.6. Unregistering the Driver

The PBX Driver must unregister before unloading, or any time it needs to break the CSTA Services Driver Interface connection. The `cdiDriverUnregister()` routine requires a single parameter, the `driverID`, returned to the PBX Driver by the `cdiDriverRegister()` routine. This routine will not block, but only signal the Cserver that the driver wishes to unregister. After the routine returns, the Cserver will send all messages sent from the driver to the Cserver via the `cdiSendToTserver()` to the appropriate clients, clear all threads from this CSTA Services Driver Interface, and unregister the driver with the Tserver by calling the `tdiDriverUnregister()` routing on behalf of the driver. The resources allocated for this interface will be released back to the Operating System, and control will be returned to the PBX Driver. No resources allocated for this interface should be accessed by the PBX Driver after the `cdiDriverUnregister()` routine completes successfully.

### 4.1.7. CSTA Services Driver Interface Monitoring

The CSTA Services Driver Interface provides two routines that the PBX Driver can access to monitor the message flow for a specific interface. The `cdiMemAllocSize()` routine provides the amount of memory (in bytes) allocated for message buffers on this interface by the PBX Driver and Cserver, and the `cdiQueueSize()` routine provides the count of messages queued to the Cserver and to the PBX Driver for this CSTA Services Driver Interface. The `driverID` that identifies the interface is the only input to these routines. The `cdiQueueSize()` and `cdiMemAllocSize()` routines return structures that describe the current state of the message queues and the bytes allocated for message buffers for both the Cserver 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 Cserver provides no form of flow control across the CSDI. There is a form of flow control in that once all the memory in a given CSDI instance is allocated to CSDI buffers via `cdiAllocBuffer()` function calls, no more CSDI buffers can be allocated (by the Cserver or PBX Driver) to be passed through the CSDI.

If it wishes to do so, the PBX Driver can also monitor the status of the TSDI using the interface-monitoring functions defined for that interface (that is, `tdiQueueSize()`, and `tdiMemAllocSize()`). The driver can pass the same `driverID` parameter to the TSDI routines as it does to the CSDI routines, since the Cserver uses the same `driverID` value to identify the CSDI interface as its corresponding TSDI interface.
4.1.8. Relationship of CSTA Services Driver Interface to TSDI

4.1.8.1. CSDI and TSDI Message Queues

Figure 3 shows the relationship of the CSDI and TSDI queues and the ROPM (Remote Operations Protocol Machine) instance that are created as a result of a single driver registration. Each such registration produces an additional instance of the queues and ROPM shown. The Cserver implements the CSDI queues and the Tserver implements the TSDI queues. Both sets of queues operate in a similar fashion and facilitate the passing of messages (that is, APDUs) between the modules on the two sides of each interface. The main differences between the two queues is the format of the messages exchanged.

The ROPM, or Remote Operations Protocol Machine, is the ROSE (Remote Operations Service Element) protocol machine. The ROPM translates ROSE service requests into ROSE APDUs (that is, application protocol data units) for transmission to the PBX Driver, and translates ROSE APDUs received from the PBX Driver into ROSE indications. The ROPM interfaces directly with the CSDI interface routines. The ROPM is not a separate thread, but runs either in the context of the Driver Receive Thread or the Tserver Receive Thread. Although not shown in Figure 3, a peer ROPM is required in the PBX Driver or in the switch. For more information about the functions of the ROPM, see [X.229].

Messages representing client requests are received by the Tserver and, after passing any security checks, are placed in the Cserver receive queue in the TSDI message format. A thread in the Cserver called the Tserver Receive Thread receives the buffers containing the messages one at a time (using the function tdiReceiveFromTserver()), and depending on the message type, either translates them into ECMA-180 encoded APDUs to be sent to the PBX Driver, or processes the messages itself and sends back replies to the Tserver. All CSTA (that is, ECMA-179) service requests and responses received by the Cserver are translated into the equivalent ECMA-180 encoded APDU and sent to the ROPM (by invoking the Invocation or Return-Result procedures) to be placed in the Driver receive queue. All ACS requests (that is, acsOpenStream), CSTA control service requests (that is, cstaGetAPICaps), Routing registration requests (that is, cstaRouteRegisterReq), and System status registration requests (that is, cstaSysStatStart) received by the Cserver are processed by the Cserver. The replies are placed on the Tserver receive queue (using the function...
tdiSendToTserver()). The PBX Driver receives ECMA-180 encoded APDUs from the Driver receive queue using the CSDI function cdiReceiveFromCSTA(), and sends them to the switch to be processed.

NOTE: Because some requests are processed in the Cserver, and some are processed in the switch/driver, the application may not always receive responses to requests in the same order as the requests were issued. This behavior does not violate TSAPI or the ECMA-179 or ECMA-180 standards.

When the switch responds to a CSTA request, or when either the switch or driver wish to initiate a request to the application (that is, system status or route request), messages representing encoded ECMA-180 APDUs are placed on the Cserver receive queue by the driver using the CSDI function cdiSendToCSTA(). The ROPM in the Cserver sends the message and sends an indication to the Driver Receive Thread and, depending on the contents of the message, the Driver Receive Thread maps the message to the TSDI format and sends it to the Tserver, or processes the message and replies to the driver. A TSDI-formatted message to be sent to the Tserver is placed on the Tserver receive queue (using the function tdiSendToTserver()) by the Cserver. A reply to the driver is placed in the Driver receive queue by invoking the Return-result operation of the ROPM in the Cserver.

4.1.8.2. PBX Driver Use of TSDI functions

The following table lists the CSDI and TSDI functions. Some of the TSDI functions can be called directly by the PBX driver while others must not be called by the PBX driver. Instead, the associated CSDI function should be called:

<table>
<thead>
<tr>
<th>CSDI Function</th>
<th>TSDI Function</th>
<th>OK To Call TSDI Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdiDriverRegister()</td>
<td>tdiDriverRegister()</td>
<td>NO</td>
</tr>
<tr>
<td>cdiDriverUnregister()</td>
<td>tdiDriverUnregister()</td>
<td>NO</td>
</tr>
<tr>
<td>cdiAllocBuffer()</td>
<td>tdiAllocBuffer()</td>
<td>NO</td>
</tr>
<tr>
<td>cdiFreeBuffer()</td>
<td>tdiFreeBuffer()</td>
<td>NO</td>
</tr>
<tr>
<td>cdiSendToCSTA()</td>
<td>tdiSendToTserver()</td>
<td>NO</td>
</tr>
<tr>
<td>cdiReceiveFromCSTA()</td>
<td>tdiReceiveFromTserver()</td>
<td>NO</td>
</tr>
<tr>
<td>cdiDriverSanity()</td>
<td>tdiDriverSanity()</td>
<td>NO</td>
</tr>
<tr>
<td>cdiQueueSize()</td>
<td>tdiQueueSize()</td>
<td>YES</td>
</tr>
<tr>
<td>cdiMemAllocSize()</td>
<td>tdiMemAllocSize()</td>
<td>YES</td>
</tr>
<tr>
<td>cdiGetCDISize()</td>
<td>tdiGetTDISize()</td>
<td>YES</td>
</tr>
<tr>
<td>(not applicable)</td>
<td>tdiSetMessageFlowControl()</td>
<td>YES</td>
</tr>
<tr>
<td>(not applicable)</td>
<td>tdiGetSessionIDInfo()</td>
<td>NO</td>
</tr>
<tr>
<td>(not applicable)</td>
<td>tdiMapInvokeID()</td>
<td>NO</td>
</tr>
<tr>
<td>(not applicable)</td>
<td>tdiLogError()</td>
<td>YES</td>
</tr>
<tr>
<td>cdiGetCSTAVersion()</td>
<td>tdiGetTservVersion</td>
<td>YES</td>
</tr>
</tbody>
</table>

4.1.9. CSTA Server Flow Control of CSDI Messages

4.1.9.1. Telephony Server Flow Control Overview

The Cserver threads and the CSDI itself behave in a way that allow the Telephony Server Flow Control mechanisms to handle any congestion that occurs in any of the CSDI or TSDI queues. For details on the Telephony Server Flow Control, see [TSDI].
If the PBX Driver does not receive messages from the *Driver receive queue* quickly enough, the queue will begin to fill with messages and eventually the *hiwater_mark* as defined in the *buffer_descriptor*, which is part of the *cdiDriverRegister()* call, will be reached. At this point, the *Tserver Receive Thread* in the Cserver will stop receiving messages from the *Cserver receive queue* in the TSDI. If the congestion persists, the *Cserver receive queue* in the TSDI will begin to fill with messages and eventually the *hiwater_mark* set for that interface will be reached and the Telephony Server Flow Control will begin to reject all new requests made by client applications. When the memory usage level at the CSDI falls back below the *lowater_mark*, the *Tserver Receive Thread* in the Cserver will once again begin to receive messages from the *Cserver receive queue* in the TSDI.

A similar form of flow control occurs in the opposite direction. When the *hiwater_mark* is reached at the TSDI, the *Driver Receive Thread* in the Cserver will stop receiving messages from the *Cserver receive queue* in the CSDI. If the congestion persists, the *Cserver receive queue* will begin to fill with messages and eventually the *hiwater_mark* set for the CSDI interface will be reached, at which point the PBX Driver should invoke its flow control mechanisms with the switch. When the memory usage level at the TSDI falls back below the *hiwater_mark*, the *Cserver Receive Thread* in the Cserver will once again begin to receive messages from the *Cserver receive queue* in the CSDI.

In the absence of prolonged congestion, no messages are discarded by the Cserver. The Cserver will always opt to allow its input queues to back up in favor of discarding messages. An excessively long period of congestion may eventually require the Cserver to discard one or more messages. In such cases, an entry is made in the error log. The message handling rate of the Cserver is a function of the processor speed, the Operating System, the thread scheduling algorithm, the server load, and the message content. Under heavy load, it is possible for the Cserver itself to become the "bottleneck" in situations where the Tserver message rate or PBX Driver message rate exceeds the Cserver's maximum message handling rate. In such cases, the same flow control mechanisms described above shall apply.

The driver should choose the *max_bytes*, *hiwater_mark*, and *lowater_mark* of the *buffer_descriptor* argument to the *cdiDriverRegister()* very carefully, as these values can greatly influence the performance of the interface. The CSTA and ROSE APDU definitions and their associated BER encoding lengths should be taken into consideration when determining these values. Note that the Cserver will use the same values of *max_bytes* and *hiwater_mark* (there is no *lowater_mark* defined for the TSDI) of the *buffer_descriptor* argument to *tdiDriverRegister()* when the Cserver registers the driver with the Tserver as those provided in the *cdiDriverRegister()* call by the PBX Driver. The maximum memory requirements for both interfaces (that is, CSDI and TSDI) are actually twice that specified in *max_bytes*. If a NULL pointer is passed in the *buffer_descriptor* argument, then the defaults specified by the TSDI shall apply to both interfaces. The default *lowater_mark* setting is half the *hiwater_mark* value. The driver can retrieve the current values of *buffer_descriptor* using the *cdiGetCDISize()* and *tdiGetTDISize()* functions.

**4.1.9.2. Selective Driver Flow Control Via tdiSetMessageFlowControl()**

The Telephony Services Driver Interface (TSDI) defines a special type of flow control mechanism which allows the driver to specify on a per request type basis when to invoke flow control. A driver wishing to use this flow control mechanism should call *tdiSetMessageFlowControl()* directly, using the same value for the *driverID* parameter as that returned by *cdiDriverRegister()* when the driver registered with the Cserver. There is no equivalent capability in the CSDI. See [TSDI] for more information on the *tdiSetMessageFlowControl()* function.

**4.2. Messaging Interface**

**4.2.1. CSDI Message Format**

CSTA request, response and event APDUs transported across the CSDI conform to the standards and protocols listed below:

- **Standard ECMA-180**: Protocol for Computer-Supported Telecommunications Applications (CSTA)
- **Recommendation X.209**: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)
The ECMA-180 standard defines the application protocol data units (APDUs) for CSTA services using the ASN.1 notation. ECMA-180 specifies that the services of CSTA are modeled as Remote Operations as described in CCITT Recommendation X.219. Recommendations X.219 and X.229 together define a model, notation, services, and protocol for the exchange of requests / responses carried out within the context of an application-association. Both the ECMA-180 and X.229 standards specify the use of the transfer syntax defined by BER (Basic Encoding Rules) for ASN.1 in CCITT Recommendation X.209. The ECMA-180 standard referenced above is the June 1992 (Blue Book) version. All the CCITT Recommendations referenced above are the 1988 (Blue Book) versions.

The recommendations contained in Recommendation X.219 are followed with one exception: X.219 requires an existing application association controlled by the service element ACSE (Association Control Service Element) defined in CCITT Rec. X.218. In the Telephony Services implementation, ACSE-like services are provided through the functions in the CSTA Services Driver Interface (CSDI), which do not conform to Recommendation X.218 but provide similar services.

The following three ROSE operation classes will be utilized, as specified in ECMA-180:

- Operation Class 2 - Asynchronous, reporting success or failure (result or error).
- Operation Class 3 - Asynchronous, reporting failure (error) only, if any.
- Operation Class 5 - Asynchronous, outcome not reported.

The message buffers transported across the CSDI must contain a single, whole, APDU as defined by the above protocols and encoding standard. Each APDU must begin at the first byte of the message buffer and must fit entirely within a single message buffer. APDUs spanning multiple sequential message buffers, or multiple APDUs in a single message buffer are not allowed. The message buffer size (that is, length specified when allocated using cdiAllocBuffer()) may be equal to or greater than the length of the actual encoded APDU contained within the buffer.

### 4.2.2. ROSE Services and Protocol

An overview of the ROSE (Remote Operations Service Element) services and associated protocol definitions (APDUs) appear in the sections that follow. The reader is advised to consult CCITT Recommendations X.219 and X.229 for the complete service and protocol definitions. Partial descriptions taken from the aforementioned standards are provided below.

#### 4.2.2.1. RO-INVOKE Service

The RO-INVOKE service is used by one ROSE-user (the invoker) to cause the invocation of an OPERATION to be performed by the other ROSE-user (the performer). This service is a non-confirmed service. The following table lists the RO-INVOKE service parameters:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Request</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation-value</td>
<td>M</td>
<td>M (=)</td>
</tr>
<tr>
<td>Operation-class</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>Argument</td>
<td>U</td>
<td>C (=)</td>
</tr>
<tr>
<td>Invoke-ID</td>
<td>M</td>
<td>M (=)</td>
</tr>
<tr>
<td>Linked-ID</td>
<td>U</td>
<td>C (=)</td>
</tr>
<tr>
<td>Priority</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

The RO-INVOKE service is utilized by the Cserver to send CSTA requests to the PBX Driver, and by the PBX Driver to send CSTA requests and events to the Cserver. The Operation-value and Argument parameters specify the CSTA service and arguments, respectively, of the desired service request. The Invoke-ID argument is needed to associate requests with responses, since the operation classes utilized by CSTA are all asynchronous. Since CSTA does not define any linked operations, the Linked-ID parameter is not used.
The following is the definition of the ROSE ROIV APDU, which is sent in response to a RO-INVOKE service invocation. Receipt of this APDU will result in a RO-INVOKE indication to the ROSE-user.

```
ROIVapdu ::= SEQUENCE {
  invokeID InvokeIDType,
  linked-ID [0] IMPLICIT InvokeIDType OPTIONAL,
  operation-value OPERATION,
  argument ANY DEFINED BY operation-value OPTIONAL }
```

-- ANY is filled by the single ASN.1 data type following
-- the key word ARGUMENT in the type definition
-- of a particular operation.

### 4.2.2.2. RO-RESULT Service

The RO-RESULT service is used by one ROSE-user to reply to a previous RO-INVOKE indication in the case of a successfully performed operation. This service is a non-confirmed service. The following table lists the RO-RESULT service parameters:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Request</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation-value</td>
<td>U</td>
<td>C (=)</td>
</tr>
<tr>
<td>Result</td>
<td>U</td>
<td>C (=)</td>
</tr>
<tr>
<td>Invoke-ID</td>
<td>M</td>
<td>M (=)</td>
</tr>
<tr>
<td>Priority</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

The RO-RESULT service is utilized by the PBX Driver to send positive responses to CSTA requests to the Cserver, and vice-versa. The Operation-value and Result parameters specify the CSTA service and results, respectively, of the desired service response. The Invoke-ID argument is needed to associate responses with requests, since the operation classes utilized by CSTA are all asynchronous.

The following is the definition of the ROSE RORS APDU, which is sent in response to a RO-RESULT service invocation:

```
RORSapdu ::= SEQUENCE {
  invokeID InvokeIDType,
  SEQUENCE {
    operation-value OPERATION,
    result ANY DEFINED BY operation-value
    -- ANY is filled by the single ASN.1 data type following
    -- the key word ARGUMENT in the type definition
    -- of a particular operation.
  } OPTIONAL }
```

### 4.2.2.3. RO-ERROR Service

The RO-ERROR service is used by one ROSE-user to reply to a previous RO_INVOKE indication in the case of an unsuccessfully performed operation. This service is a non-confirmed service. The following table lists the RO-ERROR service parameters:
The RO-ERROR service is utilized by the PBX Driver to send negative responses to CSTA requests to the Cserver, and vice-versa. The Error-value and Error-parameter specify the CSTA errors for the associated service request. The Invoke-ID argument is needed to associate negative responses with requests, since the operation classes utilized by CSTA are all asynchronous.

The following is the definition of the ROSE ROER APDU, which is sent in response to a RO-ERROR service invocation:

```
ROERapdu ::= SEQUENCE {
  invokeID InvokeIDType,
  error-value ERROR,
  parameter ANY DEFINED BY error-value OPTIONAL }
```

-- ANY is filled by the single ASN.1 data type following
-- the key word PARAMETER in the type definition
-- of a particular operation.

### 4.2.2.4. RO-REJECT-U Service

The RO-REJECT-U service is used by a ROSE-user to reject a request (RO_INVOKE indication) of the other ROSE-user if it has detected a problem. The RO-REJECT-U service may also be used by a ROSE-user to reject a reply (RO-RESULT indication, RO-ERROR indication) from the other ROSE-user. However, to avoid violating the sequencing rules of other ASEs in some application contexts, a ROSE-user may choose not to use the RO-REJECT-U service to reject replies. This service is a non-confirmed service. The following table lists the RO-REJECT-U service parameters:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Request</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reject-reason</td>
<td>M</td>
<td>M (=)</td>
</tr>
<tr>
<td>Invoke-ID</td>
<td>M</td>
<td>M (=)</td>
</tr>
<tr>
<td>Priority</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

The RO-REJECT-U service is utilized by the Cserver or PBX Driver to reject a RO-INVOKE or RO-RESULT indication due to some Cserver or PBX Driver detected problem. The Reject-reason parameter specifies the reason for rejection. The Invoke-ID argument is needed to link the rejection with its associated request or result, since the operation classes utilized by CSTA are all asynchronous.

The following is the definition of the ROSE RORJ APDU, which is sent in response to a RO-REJECT-U service invocation:

```
RORJapdu ::= SEQUENCE {
 CHOICE { invokeID CHOICE { InvokeIDType, NULL } }
  problem CHOICE { [0] IMPLICIT GeneralProblem,
                  [1] IMPLICIT InvokeProblem,
                  [2] IMPLICIT ReturnResultProblem,
                  [3] IMPLICIT ReturnErrorProblem } }
```

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Request</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reject-reason</td>
<td>M</td>
<td>M (=)</td>
</tr>
<tr>
<td>Invoke-ID</td>
<td>M</td>
<td>M (=)</td>
</tr>
<tr>
<td>Priority</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>
4.2.2.5. RO-REJECT-P Service

The RO-REJECT-P service is used to advise a ROSE-user of a problem detected by the ROSE-provider. This service is a provider-initiated service. The following table lists the RO-REJECT-P service parameters:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoke-ID</td>
<td>O</td>
</tr>
<tr>
<td>Returned-parameters</td>
<td>O</td>
</tr>
<tr>
<td>Reject-reason</td>
<td>O</td>
</tr>
</tbody>
</table>

The RO-REJECT-P service is used to report a ROSE-level problem to the ROSE-user within the Cserver or driver/switch. The RO-REJECT-P service utilizes the same ROSE APDU (that is, RORJ) as the RO-REJECT-U service (see previous section).

4.3. TSDI to ECMA-180 Message Translation

There are seven categories of CSTA services and functions defined by TSAPI:

- CSTA Control Service Functions
- CSTA Security Service Functions
- CSTA Switching Function Services
- CSTA Status Reporting Services
- CSTA Snapshot Services
- CSTA Computing Function Services
- CSTA Escape/Maintenance Services

Each of these categories has associated API calls and events, TSDI message format definitions, and ECMA-180 defined APDUs. The tables in the following sections contain the TSDI message C-structure definition, the associated ECMA-180 APDU or applicable piecepart, if any, and an indication of how one is mapped to the other for each category of CSTA services. The direction of the mapping is indicated in the mapping column with a left-arrow (that is, <-) or right-arrow (that is, ->). The left-arrow means the mapping occurs only in the direction of ECMA-180 APDU to TSDI C-structure. The right arrow means the mapping occurs only in the direction of TSDI C-structure to ECMA-180 APDU. The mapping columns also contain references (that is, "[1]" or "See note [2]") to notes immediately following the table which contain more information about the details of particular mappings. Details of the associated API calls, message class, and message type can be found in [TSAPI].

4.3.1. CSTA Control Services Functions

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

NOTE: These messages are always handled by the CSTA Server and are never sent to a driver and are listed here only for completeness; that is, every TSDI message defined in [TSDI] is listed in this document to make clear which of these messages the driver must handle and which the Cserver or Tserver handle on behalf of the driver. The PBX Driver supplies the information to be supplied in the response to the cstaGetAPICaps() function call at registration time in the driver_caps argument of the cdiDriverRegister() function call.
<table>
<thead>
<tr>
<th>TSDI C-Language Structure</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct CSTAGetAPICaps_t</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nulltype null;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} CSTAGetAPICaps_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAGetAPICapsConfEvent_t</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short alternateCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short answerCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short callCompletion;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short clearCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short clearConnection;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short conferenceCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short consultationCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short deflectCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short pickupCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short groupPickupCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short holdCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short makeCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short makePredictiveCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short queryMwi;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short queryDnd;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short queryFwd;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short queryAgentState;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short queryLastNumber;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short queryDeviceInfo;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short reconnectCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short retrieveCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short setMwi;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short setDnd;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short setFwd;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short setAgentState;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short transferCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short eventReport;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short callClearEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short conferenceEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short connectionClearEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short deliveredEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short divertedEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short establishedEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short failedEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short heldEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short networkReachedEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short originatedEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short queuedEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short retrievedEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short serviceInitiatedEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short transferredEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short callInformationEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short doNotDisturbEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short forwardingEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short messageWaitingEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short loggedOnEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short loggedOffEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short notReadyEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short readyEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short workNotReadyEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short workReadyEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short backInServiceEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short outOfServiceEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short privateEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short routeRequestEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short reRoute;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short routeSelect;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short routeUsedEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short routeEndEvent;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(continued on next page)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.3.2. 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 (Tserver) and are never sent to a driver. They are listed here only for completeness; that is, every TSDI message defined in [TSDI] is listed in this document to make clear which messages the driver must handle and which the Cserver or Tserver handle on behalf of the PBX Driver.

### CSTA Switching Function Services

CSTA Switching function services are Telephony Services that operate on calls and activate switch-related features associated with the user's desktop telephone or any other device defined by the switching domain. See [ECMA-179] for
more details on the switching function services, and see [TSAPI] for more information on the associated CSTA API C Language function calls and events.

<table>
<thead>
<tr>
<th>TSDI C-Language Structure</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct CSTAAlternateCall_t {</td>
<td>AlternateCallArgument ::= CONNECTION ConnectionDetails</td>
<td>activeCall -&gt; callsInvolved [1]</td>
</tr>
<tr>
<td>ConnectionID_t activeCall;</td>
<td>{</td>
<td>otherCall -&gt; callsInvolved [1]</td>
</tr>
<tr>
<td>ConnectionID_t otherCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}                  CSTAAlternateCall_t;</td>
<td></td>
<td>See note [2]</td>
</tr>
<tr>
<td>typedef struct CSTAAlternateCallConfEvent_t {</td>
<td>AlternateCallResult ::= CONNECTION ConnectionDetails, CSTAPrivateData</td>
<td></td>
</tr>
<tr>
<td>Nulltype null;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}                  CSTAAlternateCallConfEvent_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAAnswerCall_t {</td>
<td>AnswerCallArgument ::= CONNECTION ConnectionDetails, CSTAPrivateData</td>
<td></td>
</tr>
<tr>
<td>ConnectionID_t alertingCall;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}                  CSTAAnswerCall_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAAnswerCallConfEvent_t {</td>
<td>AnswerCallResult ::= CONNECTION ConnectionDetails, CSTAPrivateData</td>
<td></td>
</tr>
<tr>
<td>Nulltype null;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}                  CSTAAnswerCallConfEvent_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTACallCompletion_t {</td>
<td>CallCompletionArgument ::= CONNECTION ConnectionDetails</td>
<td></td>
</tr>
<tr>
<td>Feature_t feature;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>ConnectionID_t call;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}                  CSTACallCompletion_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTACallCompletionConfEvent_t {</td>
<td>CallCompletionResult ::= CONNECTION ConnectionDetails, CSTAPrivateData</td>
<td></td>
</tr>
<tr>
<td>Nulltype null;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}                  CSTACallCompletionConfEvent_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAClearCall_t {</td>
<td>ClearCallArgument ::= CONNECTION ConnectionDetails</td>
<td></td>
</tr>
<tr>
<td>ConnectionID_t call;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}                  CSTAClearCall_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAClearCallConfEvent_t {</td>
<td>ClearCallResult ::= CONNECTION ConnectionDetails, CSTAPrivateData</td>
<td></td>
</tr>
<tr>
<td>Nulltype null;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}                  CSTAClearCallConfEvent_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAClearConnection_t {</td>
<td>ClearConnectionArgument ::= CONNECTION ConnectionDetails</td>
<td></td>
</tr>
<tr>
<td>ConnectionID_t call;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}                  CSTAClearConnection_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAClearConnectionConfEvent_t {</td>
<td>ClearConnectionResult ::= CONNECTION ConnectionDetails, CSTAPrivateData</td>
<td></td>
</tr>
<tr>
<td>Nulltype null;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}                  CSTAClearConnectionConfEvent_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAConferenceCall_t {</td>
<td>ConferenceCallArgument ::= CONNECTION ConnectionDetails</td>
<td></td>
</tr>
<tr>
<td>ConnectionID_t heldCall;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>ConnectionID_t activeCall;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}                  CSTAConferenceCall_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAConferenceCallConfEvent_t {</td>
<td>ConferenceCallResult ::= CONNECTION ConnectionDetails, CSTAPrivateData</td>
<td></td>
</tr>
<tr>
<td>ConnectionID_t newCall;</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>ConnectionList_t connList;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}                  CSTAConferenceCallConfEvent_t;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
typedef struct CSTAConsultationCall_t {
    ConnectionID_t activeCall;
    DeviceID_t calledDevice;
} CSTAConsultationCall_t;

ConsultationCallArgument ::= SEQUENCE {
    existingCall ConnectionID,
    calledDirectoryNumber CalledDeviceID,
    extensions CSTACommonArguments OPTIONAL
} activeCall -> existingCall [11]
calledDevice -> calledDirectoryNumber [5]
See note [2]

typedef struct CSTAConsultationCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAConsultationCallConfEvent_t;

ConsultationCallResult ::= CHOICE {
    initiatedCall ConnectionID,
    SEQUENCE {
        initiatedCall ConnectionID,
        extensions CSTAPrivateData OPTIONAL
    }
} newCall <- initiatedCall [11]
See note [3]

typedef struct CSTADeflectCall_t {
    ConnectionID_t deflectCall;
    DeviceID_t calledDevice;
} CSTADeflectCall_t;

DivertCallArgument ::= CHOICE {
    divertInfo DivertInfo
    SEQUENCE {
        divertInfo DivertInfo,
        extensions CSTACommonArguments OPTIONAL
    }
} deflectCall -> divertInfo [6]
calledDevice -> divertInfo [6]
See note [2]

typedef struct CSTADeflectCallConfEvent_t {
    Nulltype null;
} CSTADeflectCallConfEvent_t;

DivertCallResult ::= CHOICE {
    extensions CSTAPrivateData,
    noData NULL
} See note [3]

typedef struct CSTAPickupCall_t {
    ConnectionID_t deflectCall;
    DeviceID_t calledDevice;
} CSTAPickupCall_t;

DivertCallArgument ::= CHOICE {
    divertInfo DivertInfo
    SEQUENCE {
        divertInfo DivertInfo,
        extensions CSTACommonArguments OPTIONAL
    }
} deflectCall -> divertInfo [6]
calledDevice -> divertInfo [6]
See note [2]

typedef struct CSTAPickupCallConfEvent_t {
    Nulltype null;
} CSTAPickupCallConfEvent_t;

DivertCallResult ::= CHOICE {
    extensions CSTAPrivateData,
    noData NULL
} See note [3]

typedef struct CSTAGroupPickupCall_t {
    ConnectionID_t deflectCall;
    DeviceID_t pickupDevice;
} CSTAGroupPickupCall_t;

DivertCallArgument ::= CHOICE {
    divertInfo DivertInfo
    SEQUENCE {
        divertInfo DivertInfo,
        extensions CSTACommonArguments OPTIONAL
    }
} pickupDevice -> divertInfo [6]
See note [2]

typedef struct CSTAHoldCallConfEvent_t {
    Nulltype null;
} CSTAHoldCallConfEvent_t;

HoldCallResult ::= CHOICE {
    extensions CSTAPrivateData,
    noData NULL
} See note [3]

typedef struct CSTAMakeCall_t {
    DeviceID_t callingDevice;
    DeviceID_t calledDevice;
} CSTAMakeCall_t;

MakeCallArgument ::= SEQUENCE {
    callingDevice DeviceID,
    calledDirectoryNumber CalledDeviceID,
    extensions CSTACommonArguments OPTIONAL
} callingDevice -> callingDevice [1]
calledDevice -> calledDirectoryNumber [5]
See note [2]

typedef struct CSTAMakePredictiveCall_t {
    DeviceID_t callingDevice;
    DeviceID_t calledDevice;
    AllocationState_t allocationState;
} CSTAMakePredictiveCall_t;

MakePredictiveCallArgument ::= SEQUENCE {
    callingDevice DeviceID,
    calledDirectoryNumber CalledDeviceID,
    allocation AllocationState DEFAULT callDelivered,
    extensions CSTACommonArguments OPTIONAL
} callingDevice -> callingDevice [1]
calledDevice -> calledDirectoryNumber [5]
allocationState -> allocation [2]
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct CSTAMakePredictiveCallConfEvent_t</td>
<td>ConnectionID_t newCall;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryMwi_t</td>
<td>DeviceID_t device;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryMwiConfEvent_t</td>
<td>Boolean messages;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryDnd_t</td>
<td>DeviceID_t device;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryDndConfEvent_t</td>
<td>Boolean doNotDisturb;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryFwd_t</td>
<td>DeviceID_t device;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryFwdConfEvent_t</td>
<td>ListForwardParameters_t forward;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryAgentState_t</td>
<td>DeviceID_t device;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryAgentStateConfEvent_t</td>
<td>AgentState_t agentState;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryLastNumber_t</td>
<td>DeviceID_t device;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryLastNumberConfEvent_t</td>
<td>DeviceID_t lastNumber;</td>
</tr>
<tr>
<td>typedef struct CSTAQueryDeviceInfo_t</td>
<td>DeviceID_t device;</td>
</tr>
</tbody>
</table>

- **typedef struct CSTAMakePredictiveCallConfEvent_t**
  ```
  typedef struct CSTAMakePredictiveCallConfEvent_t {
      ConnectionID_t newCall;
    } CSTAMakePredictiveCallConfEvent_t;
  ```

- **typedef struct CSTAQueryMwi_t**
  ```
  typedef struct CSTAQueryMwi_t {
      DeviceID_t device;
    } CSTAQueryMwi_t;
  ```

- **typedef struct CSTAQueryMwiConfEvent_t**
  ```
  typedef struct CSTAQueryMwiConfEvent_t {
      Boolean messages;
    } CSTAQueryMwiConfEvent_t;
  ```

- **typedef struct CSTAQueryDnd_t**
  ```
  typedef struct CSTAQueryDnd_t {
      DeviceID_t device;
    } CSTAQueryDnd_t;
  ```

- **typedef struct CSTAQueryDndConfEvent_t**
  ```
  typedef struct CSTAQueryDndConfEvent_t {
      Boolean doNotDisturb;
    } CSTAQueryDndConfEvent_t;
  ```

- **typedef struct CSTAQueryFwd_t**
  ```
  typedef struct CSTAQueryFwd_t {
      DeviceID_t device;
    } CSTAQueryFwd_t;
  ```

- **typedef struct CSTAQueryFwdConfEvent_t**
  ```
  typedef struct CSTAQueryFwdConfEvent_t {
      ListForwardParameters_t forward;
    } CSTAQueryFwdConfEvent_t;
  ```

- **typedef struct CSTAQueryAgentState_t**
  ```
  typedef struct CSTAQueryAgentState_t {
      DeviceID_t device;
    } CSTAQueryAgentState_t;
  ```

- **typedef struct CSTAQueryAgentStateConfEvent_t**
  ```
  typedef struct CSTAQueryAgentStateConfEvent_t {
      AgentState_t agentState;
    } CSTAQueryAgentStateConfEvent_t;
  ```

- **typedef struct CSTAQueryLastNumber_t**
  ```
  typedef struct CSTAQueryLastNumber_t {
      DeviceID_t device;
    } CSTAQueryLastNumber_t;
  ```

- **typedef struct CSTAQueryLastNumberConfEvent_t**
  ```
  typedef struct CSTAQueryLastNumberConfEvent_t {
      DeviceID_t lastNumber;
    } CSTAQueryLastNumberConfEvent_t;
  ```

- **typedef struct CSTAQueryDeviceInfo_t**
  ```
  typedef struct CSTAQueryDeviceInfo_t {
      DeviceID_t device;
    } CSTAQueryDeviceInfo_t;
  ```
typedef struct CSTAQueryDeviceInfoConfEvent_t {
    DeviceID_t device;
    DeviceType_t deviceType;
    DeviceClass_t deviceClass;
} CSTAQueryDeviceInfoConfEvent_t;

typedef struct CSTAReconnectCall_t {
    ConnectionID_t activeCall;
    ConnectionID_t heldCall;
} CSTAReconnectCall_t;

typedef struct CSTARetrieveCall_t {
    ConnectionID_t heldCall;
} CSTARetrieveCall_t;

typedef struct CSTASetMwi_t {
    DeviceID_t device;
    Boolean messages;
} CSTASetMwi_t;

typedef struct CSTASetDnd_t {
    DeviceID_t device;
    Boolean doNotDisturb;
} CSTASetDnd_t;

typedef struct CSTASetFwd_t {
    DeviceID_t device;
    ForwardingInfo_t forward;
} CSTASetMwi_t;

typedef struct CSTASetAgentState_t {
    DeviceID_t device;
    AgentMode_t agentMode;
    AgentID_t agentID;
    AgentGroup_t agentGroup;
    AgentPassword_t agentPassword;
} CSTASetAgentState_t;

typedef struct CSTASetAgentStateConfEvent_t {
    Nulltype null;
} CSTASetAgentStateConfEvent_t;
```c
typedef struct CSTATransferCall_t {
    ConnectionID_t heldCall;
    ConnectionID_t activeCall;
} CSTATransferCall_t;

typedef struct CSTATransferCallConfEvent_t {
    ConnectionID_t newCall;
    ConnectionList_t connList;
} CSTATransferCallConfEvent_t;

typedef struct CSTAUniversalFailureConfEvent_t {
    CSTAUniversalFailure_t error;
} CSTAUniversalFailureConfEvent_t;

TransferCallArgument ::= 
  CHOICE
  {transferInfo ConnectionDetails } 
  SEQUENCE
  {transferInfo ConnectionDetails, 
    extensions CSTACommonArguments OPTIONAL } )
  See note [2]

TransferCallResult ::= 
  SEQUENCE
  {transferredCall ConnectionID, 
    connections ConnectionList OPTIONAL, 
    extensions CSTAPrivateData OPTIONAL } 
  See note [3]

UniversalFailure ::= 
  CHOICE
  {operationalErrors [1] IMPLICIT Operations, 
    stateErrors [2] IMPLICIT StateIncompatibility, 
    systemResourceErrors [3] IMPLICIT SystemResourceAvailability, 
    performanceErrors [5] IMPLICIT PerformanceManagement, 
    securityErrors [6] IMPLICIT SecurityError, 
    unspecifiedErrors [7] IMPLICIT NULL, 
    nonStandardErrors CSTAPrivateData } 
  See note [13]
```

Note [1] See section {4.4.1.} for details on the mapping of DeviceID.
Note [2] See section {4.4.8.} for details on the mapping of CSTACommonArguments.
Note [3] See section {4.4.8.} for details on the mapping of CSTAPrivateData.
Note [4] See section {4.4.3.} for details on the mapping of ConnectionList.
Note [5] See section {4.4.1.} for details on the mapping of CalledDeviceID.
Note [7] See section {4.4.5.} for details on the mapping of QueryDeviceFeature.
Note [8] See section {4.4.5.} for details on the mapping of QueryDeviceInfo.
Note [9] See section {4.4.5.} for details on the mapping of SetDeviceFeature.
Note [10] See section {4.4.5.} for details on the mapping of ConnectionDetails.
Note [11] See section {4.4.2.} for details on the mapping of ConnectionID.
Note [12] See section {4.4.5.} for details on the mapping of FeatureInfo.
Note [13] See section {4.4.6.} for details on the mapping of UniversalFailure.
4.3.4. CSTA Status Reporting Services

CSTA 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 (that is, a desktop telephone). See [ECMA-179] for more details on the status reporting services, and see [TSAPI] for more information on the associated CSTA API C Language function calls and events.

4.3.4.1. Status Reporting Functions

Status Reporting functions allow the application to issue requests to set, change, or cancel a monitor on a CSTA object (that is, CSTA device or call). Three types of monitoring are provided: device, call, or calls via device. See [TSAPI] and [ECMA-179] for more details on these services and associated functions and events.

NOTE 1: A CSTAMonitorEndedEvent_t event is generated by Cserver in response to receipt of a Monitor Stop service request from the switch/driver. The Cserver will positively acknowledge the request and send the CSTAMonitorEndedEvent event to the client application associated with the monitor. The cause member of CSTAMonitorEndedEvent_t is always set to EC_NONE.

NOTE 2: The PBX Driver must terminate a monitor by sending a Monitor Stop service request to the Cserver if the object being monitored expires, which can happen for a call, or if the object leaves a CSTA sub-domain, which can happen for all objects. Failure to do so will unnecessarily tie up resources in the Cserver and will eventually cause failures when any associated monitorCrossRefIDs are reused.

<table>
<thead>
<tr>
<th>TSDI C-Language Structure</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct CSTAMonitorDevice_t</td>
<td>MonitorStartArgument ::= SEQUENCE</td>
<td>deviceID -&gt; monitorObject [1]</td>
</tr>
<tr>
<td></td>
<td>(monitorObject MonitorObject,</td>
<td>monitorFilter -&gt; monitorFilter [2]</td>
</tr>
<tr>
<td></td>
<td>monitorFilter MonitorFilter OPTIONAL,</td>
<td>See note [3]</td>
</tr>
<tr>
<td></td>
<td>monitorType MonitorType OPTIONAL,</td>
<td>See note [4]</td>
</tr>
<tr>
<td></td>
<td>extensions CSTACommonArguments OPTIONAL)</td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAMonitorCall_t</td>
<td>MonitorStartArgument ::= SEQUENCE</td>
<td>call -&gt; monitorObject [1]</td>
</tr>
<tr>
<td></td>
<td>(monitorObject MonitorObject,</td>
<td>monitorFilter -&gt; monitorFilter [2]</td>
</tr>
<tr>
<td></td>
<td>monitorFilter MonitorFilter OPTIONAL,</td>
<td>See note [3]</td>
</tr>
<tr>
<td></td>
<td>monitorType MonitorType OPTIONAL,</td>
<td>See note [4]</td>
</tr>
<tr>
<td></td>
<td>extensions CSTACommonArguments OPTIONAL)</td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAMonitorCallsViaDevice_t</td>
<td>MonitorStartArgument ::= SEQUENCE</td>
<td>deviceID -&gt; monitorObject [1]</td>
</tr>
<tr>
<td></td>
<td>(monitorObject MonitorObject,</td>
<td>monitorFilter -&gt; monitorFilter [2]</td>
</tr>
<tr>
<td></td>
<td>monitorFilter MonitorFilter OPTIONAL,</td>
<td>See note [3]</td>
</tr>
<tr>
<td></td>
<td>monitorType MonitorType OPTIONAL,</td>
<td>See note [4]</td>
</tr>
<tr>
<td></td>
<td>extensions CSTACommonArguments OPTIONAL)</td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAMonitorConfEvent_t</td>
<td>MonitorStartResult ::= SEQUENCE</td>
<td>monitorCrossRefID &lt;- crossRefIdentifier [5]</td>
</tr>
<tr>
<td></td>
<td>(crossRefIdentifier MonitorCrossRefID,</td>
<td>monitorFilter &lt;- monitorFilter [2]</td>
</tr>
<tr>
<td></td>
<td>monitorFilter MonitorFilter OPTIONAL,</td>
<td>See note [6]</td>
</tr>
<tr>
<td></td>
<td>monitorType MonitorType OPTIONAL,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extensions CSTAPrivateData OPTIONAL)</td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTAChangeMonitorFilter_t</td>
<td>ChangeMonitorFilterArgument ::= SEQUENCE</td>
<td>monitorCrossRefID -&gt; monitorCrossRefID [5]</td>
</tr>
<tr>
<td></td>
<td>(monitorCrossRefID MonitorCrossRefID,</td>
<td>monitorFilter -&gt; filterList [2]</td>
</tr>
<tr>
<td></td>
<td>filterlist MonitorFilter OPTIONAL,</td>
<td>See note [4]</td>
</tr>
<tr>
<td></td>
<td>extensions CSTAPrivateData OPTIONAL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[0] IMPLICIT MonitorFilter,</td>
<td>See note [6]</td>
</tr>
<tr>
<td></td>
<td>extensions MonitorFilter OPTIONAL,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>filterList MonitorFilter OPTIONAL,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extensions CSTAPrivateData OPTIONAL)</td>
<td></td>
</tr>
</tbody>
</table>
### 4.3.4.2. Call Event Reports

Call Event reports are status reports sent by the switch/driver to the application in response to a change of state of one or more CSTA calls or connections in the switching sub-domain. Each Call Event report may contain a parameter that summarizes the local connection state as perceived from a device that is the monitored object. Every Call Event report may contain one cause code or field that adds meaning or clarifies the basic meaning of the event report. The defined Call Event reports may apply to a single connection, multiple connections within a single call, or multiple connections within multiple calls.

<table>
<thead>
<tr>
<th>TSDI C-Language Structure</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct CSTACallClearedEvent_t { \n  ConnectionID_t clearedCall; \n  LocalConnectionState_t localConnectionInfo; \n} CSTACallClearedEvent_t;</td>
<td>CallClearedEventInfo ::= SEQUENCE { \n  clearedCall ConnectionID, \n  localConnectionInfo LocalConnectionState OPTIONAL, \n  cause EventCause OPTIONAL \n}</td>
<td>clearedCall &lt;- clearedCall [1] \nlocalConnectionInfo &lt;- localConnectionInfo [2] \ncause &lt;- cause [3]</td>
</tr>
<tr>
<td>typedef struct CSTAConferencedEvent_t { \n  ConnectionID_t primaryOldCall; \n  ConnectionID_t secondaryOldCall; \n  SubjectDeviceID_t contrController; \n  SubjectDeviceID_t addedParty; \n  ConnectionList_t conferenceConnections; \n  LocalConnectionState_t localConnectionInfo; \n} CSTAConferencedEvent_t;</td>
<td>ConferencedEventInfo ::= SEQUENCE { \n  primaryOldCall ConnectionID, \n  secondaryOldCall ConnectionID OPTIONAL, \n  contrController SubjectDeviceID, \n  addedParty SubjectDeviceID, \n  conferenceConnections ConnectionList OPTIONAL, \n  localConnectionInfo LocalConnectionState OPTIONAL, \n  cause EventCause OPTIONAL \n}</td>
<td>primaryOldCall &lt;- primaryOldCall [1] \nsecondaryOldCall &lt;- secondaryOldCall [1] \ncontrController &lt;- contrController [4] \naddedParty &lt;- addedParty [4] \nconferenceConnections &lt;- conferenceConnections [5] \nlocalConnectionInfo &lt;- localConnectionInfo [2] \ncause &lt;- cause [3]</td>
</tr>
<tr>
<td>typedef struct CSTACallConnectionClearedEvent_t { \n  ConnectionID_t droppedConnection; \n  SubjectDeviceID_t releasingDevice; \n  LocalConnectionState_t localConnectionInfo; \n} CSTACallConnectionClearedEvent_t;</td>
<td>CallClearedEventInfo ::= SEQUENCE { \n  droppedConnection ConnectionID, \n  releasingDevice SubjectDeviceID, \n  localConnectionInfo LocalConnectionState OPTIONAL, \n  cause EventCause OPTIONAL \n}</td>
<td>droppedConnection &lt;- droppedConnection [1] \nreleasingDevice &lt;- releasingDevice [4] \nlocalConnectionInfo &lt;- localConnectionInfo [2] \ncause &lt;- cause [3]</td>
</tr>
</tbody>
</table>
### CSTA Services PBX Driver Interface Specification

**OriginatedEventInfo**

```c
typedef struct CSTAOriginatedEvent_t {
    ConnectionID_t        connection;   // [1]
    SubjectDeviceID_t     alertingDevice; // [4]
    CallingDeviceID_t     callingDevice; // [6]
    CalledDeviceID_t      calledDevice; // [7]
    RedirectionDeviceID_t lastRedirectionDevice; // [8]
    LocalConnectionState_t localConnectionInfo; // [2]
    CSTAEventCause_t      cause;         // [3]
} CSTAOriginatedEvent_t;
```

**NetworkReachedEventInfo**

```c
typedef struct CSTANetworkReachedEvent_t {
    connection ConnectionID OPTIONAL; // [1]
    alertingDevice SubjectDeviceID OPTIONAL; // [4]
    callingDevice CallingDeviceID OPTIONAL; // [6]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    lastRedirectionDevice RedirectionDeviceID OPTIONAL; // [8]
    localConnectionInfo LocalConnectionState OPTIONAL; // [2]
    cause EventCause OPTIONAL; // [3]
} CSTANetworkReachedEvent_t;
```

**HeldEventInfo**

```c
typedef struct CSTAHeldEvent_t {
    connection ConnectionID OPTIONAL; // [1]
    alertingDevice SubjectDeviceID OPTIONAL; // [4]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    trunkUsed SubjectDeviceID OPTIONAL; // [8]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    callingDevice CallingDeviceID OPTIONAL; // [6]
    localConnectionInfo LocalConnectionState OPTIONAL; // [2]
    cause EventCause OPTIONAL; // [3]
} CSTAHeldEvent_t;
```

**FailedEventInfo**

```c
typedef struct CSTAFailedEvent_t {
    connection ConnectionID OPTIONAL; // [1]
    alertingDevice SubjectDeviceID OPTIONAL; // [4]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    trunkUsed SubjectDeviceID OPTIONAL; // [8]
    failingDevice SubjectDeviceID OPTIONAL; // [4]
    failedDevice SubjectDeviceID OPTIONAL; // [4]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    callingDevice CallingDeviceID OPTIONAL; // [6]
    localConnectionInfo LocalConnectionState OPTIONAL; // [2]
    cause EventCause OPTIONAL; // [3]
} CSTAFailedEvent_t;
```

**EstablishedEventInfo**

```c
typedef struct CSTAEstablishedEvent_t {
    establishedConnection ConnectionID OPTIONAL; // [1]
    answeringDevice SubjectDeviceID OPTIONAL; // [4]
    callingDevice CalledDeviceID OPTIONAL; // [6]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    newDestination CalledDeviceID OPTIONAL; // [4]
    answeringDevice SubjectDeviceID OPTIONAL; // [4]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    callingDevice CallingDeviceID OPTIONAL; // [6]
    localConnectionInfo LocalConnectionState OPTIONAL; // [2]
    cause EventCause OPTIONAL; // [3]
} CSTAEstablishedEvent_t;
```

**DivertedEventInfo**

```c
typedef struct CSTADivertedEvent_t {
    connection ConnectionID OPTIONAL; // [1]
    divertingDevice SubjectDeviceID OPTIONAL; // [4]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    newDestination CalledDeviceID OPTIONAL; // [7]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    callingDevice CallingDeviceID OPTIONAL; // [6]
    localConnectionInfo LocalConnectionState OPTIONAL; // [2]
    cause EventCause OPTIONAL; // [3]
} CSTADivertedEvent_t;
```

**DeliveredEventInfo**

```c
typedef struct CSTADeliveredEvent_t {
    connection ConnectionID OPTIONAL; // [1]
    callingDevice CalledDeviceID OPTIONAL; // [6]
    divertingDevice SubjectDeviceID OPTIONAL; // [4]
    lastRedirectionDevice RedirectionDeviceID OPTIONAL; // [8]
    calledDevice CalledDeviceID OPTIONAL; // [7]
    callingDevice CallingDeviceID OPTIONAL; // [6]
    alertingDevice SubjectDeviceID OPTIONAL; // [4]
    localConnectionInfo LocalConnectionState OPTIONAL; // [2]
    cause EventCause OPTIONAL; // [3]
} CSTADeliveredEvent_t;
```

**AlertingEventInfo**

```c
typedef struct CSTAAlertingEvent_t {
    alertingDevice SubjectDeviceID OPTIONAL; // [4]
    callingDevice CalledDeviceID OPTIONAL; // [6]
    divertingDevice SubjectDeviceID OPTIONAL; // [4]
    lastRedirectionDevice RedirectionDeviceID OPTIONAL; // [8]
    callingDevice CalledDeviceID OPTIONAL; // [6]
    alertingDevice SubjectDeviceID OPTIONAL; // [4]
    localConnectionInfo LocalConnectionState OPTIONAL; // [2]
    cause EventCause OPTIONAL; // [3]
} CSTAAlertingEvent_t;
```
typedef struct CSTAQueuedEvent_t {
    ConnectionID_t queuedConnection;
    SubjectDeviceID_t queue;
    CallingDeviceID_t callingDevice;
    CalledDeviceID_t calledDevice;
    RedirectionDeviceID_t lastRedirectionDevice;
    short numberQueued;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAQueuedEvent_t;

QueuedEventInfo ::= SEQUENCE {
    queuedConnection ConnectionID,    
    queue SubjectDeviceID,          
    callingDevice CallingDeviceID,  
    calledDevice CalledDeviceID,    
    lastRedirectionDevice RedirectionDeviceID,  
    numberQueued NoOfCallsInQueue OPTIONAL,  
    localConnectionInfo LocalConnectionState OPTIONAL,  
    cause EventCause OPTIONAL
}

QueuedEventInfo ::= SEQUENCE {
    queuedConnection ConnectionID,    
    queue SubjectDeviceID,          
    callingDevice CallingDeviceID,  
    calledDevice CalledDeviceID,    
    lastRedirectionDevice RedirectionDeviceID,  
    numberQueued NoOfCallsInQueue OPTIONAL,  
    localConnectionInfo LocalConnectionState OPTIONAL,  
    cause EventCause OPTIONAL
}

typedef struct CSTARetrievedEvent_t {
    ConnectionID_t retrievedConnection;
    SubjectDeviceID_t retrievingDevice;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTARetrievedEvent_t;

RetrievedEventInfo ::= SEQUENCE {
    retrievedConnection ConnectionID,    
    retrievingDevice SubjectDeviceID,  
    localConnectionInfo LocalConnectionState OPTIONAL,  
    cause EventCause OPTIONAL
}

RetrievedEventInfo ::= SEQUENCE {
    retrievedConnection ConnectionID,    
    retrievingDevice SubjectDeviceID,  
    localConnectionInfo LocalConnectionState OPTIONAL,  
    cause EventCause OPTIONAL
}

typedef struct CSTAServiceInitiatedEvent_t {
    ConnectionID_t initiatedConnection;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTAServiceInitiatedEvent_t;

ServiceInitiatedEventInfo ::= SEQUENCE {
    initiatedConnection ConnectionID,    
    localConnectionInfo LocalConnectionState OPTIONAL,  
    cause EventCause OPTIONAL
}

ServiceInitiatedEventInfo ::= SEQUENCE {
    initiatedConnection ConnectionID,    
    localConnectionInfo LocalConnectionState OPTIONAL,  
    cause EventCause OPTIONAL
}

typedef struct CSTATransferredEvent_t {
    ConnectionID_t primaryOldCall;
    ConnectionID_t secondaryOldCall;
    SubjectDeviceID_t transferringDevice;
    SubjectDeviceID_t transferredDevice;
    ConnectionList_t transferredConnections;
    LocalConnectionState_t localConnectionInfo;
    CSTAEventCause_t cause;
} CSTATransferredEvent_t;

TransferredEventInfo ::= SEQUENCE {
    primaryOldCall ConnectionID,  
    secondaryOldCall ConnectionID OPTIONAL,  
    transferringDevice SubjectDeviceID,  
    transferredDevice SubjectDeviceID,  
    transferredConnections ConnectionList OPTIONAL,  
    localConnectionInfo LocalConnectionState OPTIONAL,  
    cause EventCause OPTIONAL
}

TransferredEventInfo ::= SEQUENCE {
    primaryOldCall ConnectionID,  
    secondaryOldCall ConnectionID OPTIONAL,  
    transferringDevice SubjectDeviceID,  
    transferredDevice SubjectDeviceID,  
    transferredConnections ConnectionList OPTIONAL,  
    localConnectionInfo LocalConnectionState OPTIONAL,  
    cause EventCause OPTIONAL
}

Note [1] See section {4.4.2.} for details on the mapping of ConnectionID.
Note [2] See section {4.4.3.} for details on the mapping of LocalConnectionState.
Note [3] See section {4.4.6.} for details on the mapping of EventCause.
Note [4] See section {4.4.1.} for details on the mapping of SubjectDeviceID.
Note [5] See section {4.4.3.} for details on the mapping of ConnectionList.
Note [6] See section {4.4.1.} for details on the mapping of CallingDeviceID.
Note [7] See section {4.4.1.} for details on the mapping of CalledDeviceID.
Note [8] See section {4.4.1.} for details on the mapping of RedirectionDeviceID.
Note [9] See section {4.4.5.} for details on the mapping of NoOfCallsInQueue.

4.3.4.3. Feature Event Reports

Feature Event reports are status reports sent by the switch/driver to the application in response to a change in feature state of a call or device in the CSTA network. Like Call Event reports, each Feature Event report indicates the new state that the feature enters regardless of any previous state.
### CSTA Services PBX Driver Interface Specification

**typedef struct CSTADoNotDisturbEvent_t**

- SubjectDeviceID_t device;
- Boolean doNotDisturbOn;

**CSTADoNotDisturbEvent_t**

**DoNotDisturbEventInfo** :=

- device SubjectDeviceID_t,
- doNotDisturbOn BOOLEAN

**ForwardingEventInfo** :=

- device SubjectDeviceID_t,
- forwardingInformation ForwardParameter

**MessageWaitingEventInfo** :=

- deviceCalledDeviceID_t,
- invokingDevice SubjectDeviceID_t,
- messageWaitingOn BOOLEAN

**Note** [1] See section {4.4.2.} for details on the mapping of ConnectionID.

**Note** [2] See section {4.4.1.} for details on the mapping of SubjectDeviceID.

**Note** [3] See section {4.4.5.} for details on the mapping of AccountInfo.

**Note** [4] See section {4.4.5.} for details on the mapping of AuthCode.

**Note** [5] See section {4.4.5.} for details on the mapping of ForwardParameter.

**Note** [6] See section {4.4.1.} for details on the mapping of CalledDeviceID.

#### 4.3.4.4. Agent State Event Reports

Agent State Event reports are status reports sent by the switch/driver to the application in response to a change in state of an agent in the CSTA network. Like Call Event reports, each Agent State Event report indicates the new state that the agent enters regardless of any previous state. Every Agent State Event report includes the identifier of the agent device to which the event report applies.

<table>
<thead>
<tr>
<th>TSDI C-Language Structure</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
</table>
| **typedef struct CSTALoggedOnEvent_t**
  (SubjectDeviceID_t agentDevice;
   AgentID_t agentID;
   AgentGroup_t agentGroup;
   AgentPassword_t password;
) CSTALoggedOnEvent_t; | **LoggedOnEventInfo** :=
  (agentDevice SubjectDeviceID_t,
   agentID [10] IMPLICIT AgentID OPTIONAL,
   agentGroup AgentGroup OPTIONAL,
| agentDevice <- agentDevice [1];
  agentID <- agentID [2];
  agentGroup <- agentGroup [3];
  password <- password [4]; |
| **typedef struct CSTALoggedOffEvent_t**
  (SubjectDeviceID_t agentDevice;
   AgentID_t agentID;
   AgentGroup_t agentGroup;
) CSTALoggedOffEvent_t; | **LoggedOffEventInfo** :=
  (agentDevice SubjectDeviceID_t,
   agentID [10] IMPLICIT AgentID OPTIONAL,
   agentGroup AgentGroup OPTIONAL)
| agentDevice <- agentDevice [1];
  agentID <- agentID [2];
  agentGroup <- agentGroup [3]; |
| **typedef struct CSTANotReadyEvent_t**
  (SubjectDeviceID_t agentDevice;
   AgentID_t agentID;
) CSTANotReadyEvent_t; | **NotReadyEventInfo** :=
  (agentDevice SubjectDeviceID_t,
   agentID [10] IMPLICIT AgentID OPTIONAL)
| agentDevice <- agentDevice [1];
  agentID <- agentID [2]; |
| **typedef struct CSTAReadyEvent_t**
  (SubjectDeviceID_t agentDevice;
   AgentID_t agentID;
) CSTAReadyEvent_t; | **ReadyEventInfo** :=
  (agentDevice SubjectDeviceID_t,
   agentID [10] IMPLICIT AgentID OPTIONAL)
| agentDevice <- agentDevice [1];
  agentID <- agentID [2]; |
| **typedef struct CSTAWorkNotReadyEvent_t**
  (SubjectDeviceID_t agentDevice;
   AgentID_t agentID;
) CSTAWorkNotReadyEvent_t; | **WorkNotReadyEventInfo** :=
  (agentDevice SubjectDeviceID_t,
   agentID [10] IMPLICIT AgentID OPTIONAL)
| agentDevice <- agentDevice [1];
  agentID <- agentID [2]; |
typedef struct CSTAWorkReadyEvent_t
{
    SubjectDeviceID_t agentDevice;
    AgentID_t agentID;
} CSTAWorkReadyEvent_t;

typedef struct CSTAWorkReadyEvent_t
{
    SubjectDeviceID_t agentDevice;
    AgentID_t agentID;
} CSTAWorkReadyEvent_t;

typedef struct CSTAWorkReadyEvent_t
{
    SubjectDeviceID_t agentDevice;
    AgentID_t agentID;
} CSTAWorkReadyEvent_t;

typedef struct CSTAWorkReadyEvent_t
{
    SubjectDeviceID_t agentDevice;
    AgentID_t agentID;
} CSTAWorkReadyEvent_t;

typedef struct CSTAWorkReadyEvent_t
{
    SubjectDeviceID_t agentDevice;
    AgentID_t agentID;
} CSTAWorkReadyEvent_t;

Note [1] See section [4.4.1.] for details on the mapping of SubjectDeviceID.
Note [2] See section [4.4.5.] for details on the mapping of AgentID.

4.3.5. CSTA Snapshot Services

CSTA Snapshot 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.

<table>
<thead>
<tr>
<th>TSDI C-Language Structure</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
</table>
| typedef struct CSTASnapshotDevice_t
{
    DeviceID_t snapshotObject;
} CSTASnapshotDevice_t; |
| SnapshoDeviceArgument ::= CHOICE
  {snapshotObject DeviceID,
    sequence
    {snapshotObject DeviceID,
      extensions CSTACommonArguments OPTIONAL}} |
| snapshotObject -> snapshotObject [1]
See note [1] |
| typedef struct CSTASnapshotDeviceConfEvent_t
{
    CSTASnapshotDeviceData_t snapshotData;
} CSTASnapshotDeviceConfEvent_t; |
| SnapshotDeviceResult ::= CHOICE
  {snapshotData SnapshotDeviceData,
    sequence
    {snapshotData SnapshotDeviceData,
      extensions CSTAPrivateData OPTIONAL}} |
| snapshotData <- snapshotData |
| typedef struct CSTASnapshotCall_t
{
    ConnectionID_t snapshotObject;
} CSTASnapshotCall_t; |
| SnapshoDeviceArgument ::= CHOICE
  {snapshotObject ConnectionID,
    sequence
    {snapshotObject ConnectionID,
      extensions CSTACommonArguments OPTIONAL}} |
| snapshotObject -> snapshotObject [1]
See note [1] |
| typedef struct CSTASnapshotCallConfEvent_t
{
    CSTASnapshotCallData_t snapshotData;
} CSTASnapshotCallConfEvent_t; |
| SnapshotDeviceResult ::= CHOICE
  {snapshotData SnapshotCallData,
    sequence
    {snapshotData SnapshotCallData,
      extensions CSTAPrivateData OPTIONAL}} |
| snapshotData <- snapshotData |

Note [1] See section [4.4.8.] for details on the mapping of CSTACommonArguments.
Note [4] See section [4.4.2.] for details on the mapping of ConnectionID.

4.3.6. CSTA Computing Function Services

CSTA Computing Function 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, and see [TSAPI] for more information on the associated CSTA API C Language function calls and events.
4.3.6.1. Routing Registration Functions and Events

This section describes 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 (that is, route point on the switch) or as a default routing server. The Cserver permits as many simultaneous routing registrations as the number of simultaneous clients that the Telephony Server supports.

NOTE: These messages are always handled by the CSTA Server (Cserver) and are never sent to a driver that has registered with the Cserver. They are listed here only for completeness; that is, every TSDI message C structure [TSDI] is listed in this document to make clear which messages the driver must handle and which the Cserver or Tserver handle on behalf of the PBX Driver.

4.3.6.2. Routing Functions and Events

This section defines the CSTA call routing services that can be utilized for application-based call routing within the switching domain. Calls that are routed using these services are queued at the routing device until the application provides a destination for the call or a 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.

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; that is, the application. The information contained in the route request from the switch/driver determines the application to which the CSTARouteRequestEvent will be sent. The currentRoute device within the request is used to find the application that has registered to be the routing server for this same device. If the application is found, then the CSTARouteRequestEvent is sent to it. If it is not found, then the event is sent to the default routing server application, if one exists. Otherwise, no application receives the CSTARouteRequestEvent and a route end request is sent to the switch/driver. When a routing dialog is established, any subsequent re-route, route used, or route end requests within the same routing dialog sent from the switch/driver are sent to the same routing server 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 (that is, 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 from the one provided previously) using the **CSTARerouteEvent**.
4.3.6.3 Register Request ID vs. Routing Cross Reference ID

The routing services described in this document use two 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 Route Request. The `routingCrossRefID` specified in the route request event will be valid for the duration of the call routing dialog or until a route end request is sent by either the driver/switch or the application.

The `routeRegisterReqID` is generated by the Cserver and the `routingCrossRefID` is generated by the switch/driver. A switch/driver that registers with the Cserver never handles the `routeRegisterReqID`.

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

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct CSTARouteRequestEvent_t</td>
<td>RouteRequestArgument ::=</td>
<td>See note [1]</td>
</tr>
<tr>
<td>{</td>
<td>crossRefIdentifier RoutingCrossRefID;</td>
<td>routingCrossRefID &lt;-&gt; crossRefIdentifier [2]</td>
</tr>
<tr>
<td>routeRegisterReqID;</td>
<td>currentRoute CalledDeviceID;</td>
<td>currentRequest &lt;-&gt; currentRoute [3]</td>
</tr>
<tr>
<td>RoutingCrossRefID_t</td>
<td>callingDevice IncludedDeviceID;</td>
<td>callingDevice &lt;-&gt; callingDevice [4]</td>
</tr>
<tr>
<td>DeviceID_t</td>
<td>routeCall ConnectionID;</td>
<td>routeCall &lt;-&gt; routeCall [5]</td>
</tr>
<tr>
<td>DeviceID_t</td>
<td>routeSetAlgorithm SelectValue;</td>
<td>routeSetAlgorithm &lt;-&gt; routeSetAlgorithm [6]</td>
</tr>
<tr>
<td>ConnectionID_t</td>
<td>priority PriorityValue OPTIONAL;</td>
<td>priority &lt;-&gt; PriorityValue [7]</td>
</tr>
<tr>
<td>SelectValue_t</td>
<td>setupInformation SetupValues OPTIONAL;</td>
<td>setupInformation &lt;-&gt; setupInformation [8]</td>
</tr>
<tr>
<td>Boolean</td>
<td>extensions CSTACommonArguments OPTIONAL;</td>
<td>See note [9]</td>
</tr>
<tr>
<td>SetUpValues_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTARouteRequestEvent_t</td>
<td>ReRouteRequestArgument ::=</td>
<td>See note [1]</td>
</tr>
<tr>
<td>{</td>
<td>crossRefIdentifier RoutingCrossRefID;</td>
<td>routingCrossRefID &lt;-&gt; crossRefIdentifier [2]</td>
</tr>
<tr>
<td>routeRegisterReqID;</td>
<td>extensions CSTACommonArguments OPTIONAL;</td>
<td>See note [9]</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTARouteSelectRequest_t</td>
<td>RouteSelectRequestArgument ::=</td>
<td>See note [1]</td>
</tr>
<tr>
<td>{</td>
<td>crossRefIdentifier RoutingCrossRefID;</td>
<td>routingCrossRefID &lt;-&gt; crossRefIdentifier [2]</td>
</tr>
<tr>
<td>routeRegisterReqID;</td>
<td>remainRetry RetryValue OPTIONAL;</td>
<td>remainRetry &lt;-&gt; remainRetry [10]</td>
</tr>
<tr>
<td>RoutingCrossRefID_t</td>
<td>setupInformation SetupValues OPTIONAL;</td>
<td>setupInformation &lt;-&gt; setupInformation [8]</td>
</tr>
<tr>
<td>DeviceID_t</td>
<td>routeUsedReq RouteUsedFlag OPTIONAL;</td>
<td>routeUsedReq &lt;-&gt; routeUsedReq [11]</td>
</tr>
<tr>
<td>RouteSelectRequest_t</td>
<td>extensions CSTACommonArguments OPTIONAL;</td>
<td>See note [9]</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct CSTARouteUsedEvent_t</td>
<td>RouteUsedArgument ::=</td>
<td>See note [1]</td>
</tr>
<tr>
<td>{</td>
<td>crossRefIdentifier RoutingCrossRefID;</td>
<td>routingCrossRefID &lt;-&gt; crossRefIdentifier [2]</td>
</tr>
<tr>
<td>routeRegisterReqID;</td>
<td>routeUsedCalledDeviceID;</td>
<td>routeUsed &lt;-&gt; routeUsedCalledDeviceID [3]</td>
</tr>
<tr>
<td>DeviceID_t</td>
<td>callingDevice IncludedDeviceID;</td>
<td>callingDevice &lt;-&gt; callingDevice [4]</td>
</tr>
<tr>
<td>DeviceID_t</td>
<td>domain DomainValue OPTIONAL;</td>
<td>domain &lt;-&gt; DomainValue [12]</td>
</tr>
<tr>
<td>Boolean</td>
<td>extensions CSTACommonArguments OPTIONAL;</td>
<td>See note [9]</td>
</tr>
<tr>
<td>SetUpValues_t</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See note [1]
typedef struct CSTARouteEndRequest_t {
    RouteRegisterReqID_t routeRegisterReqID;
    RoutingCrossRefID_t routingCrossRefID;
    CSTAUniversalFailure_t errorValue;
} CSTARouteEndRequest_t;

RouteEndRequestArgument ::= SEQUENCE {
    crossRefIdentifier RoutingCrossRefID,
    errorValue UniversalFailure OPTIONAL,
    extensions CSTACommonArguments OPTIONAL
}

See note [1]

routingCrossRefID -> crossRefIdentifier [2]

errorValue -> errorValue [13]

See note [9]

Note [1] See section { 4.4.5. } for details on the mapping of RouteRegisterReqID_t.
Note [2] See section { 4.4.5. } for details on the mapping of RoutingCrossRefID.
Note [3] See section { 4.4.1. } for details on the mapping of CalledDeviceID.
Note [4] See section { 4.4.1. } for details on the mapping of CallingDeviceID.
Note [5] See section { 4.4.2. } for details on the mapping of ConnectionID.
Note [6] See section { 4.4.5. } for details on the mapping of SelectValue.
Note [7] See section { 4.4.5. } for details on the mapping of PriorityValue.
Note [8] See section { 4.4.5. } for details on the mapping of SetupValues.
Note [9] See section { 4.4.8. } for details on the mapping of CSTACommonArguments.
Note [10] See section { 4.4.5. } for details on the mapping of RetryValue.
Note [11] See section { 4.4.5. } for details on the mapping of RouteUsedFlag.
Note [12] See section { 4.4.5. } for details on the mapping of DomainValue.

4.3.7. CSTA Escape/Maintenance Services

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

- the device status maintenance events that provide status information for device objects and
- bidirectional system status maintenance services that provide information on the overall status of the system.

The device status events inform the application when a monitored device is placed in or out of service. When a device object is placed out of service, the application will be limited to monitoring the device (for example, cstaMonitorDevice() or cstaDevSnapshotReq()). 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; that is, the monitorCrossRefID. The driver must enforce this limitation.

4.3.7.1. Escape Services: Application as Client

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

NOTE: The CSTAPrivateEvent_t type event is not supported in the Cserver because ECMA-179 only defines private events that are associated with a monitored object. The CSTAPrivateStatusEvent_t type event is sent to the application in response to receiving a (ECMA-179) private event from the switch/driver.
typedef struct CSTAEscapeSvcReqConfEvent_t
{
    Нультипе null;
} CSTAEscapeSvcReqConfEvent_t;

typedef struct CSTAPrivateEvent_t
{
    Нультипе null;
} CSTAPrivateEvent_t;

typedef struct CSTAPrivateStatusEvent_t
{
    Нультипе null;
} CSTAPrivateStatusEvent_t;

EscapeServiceResult ::= CHOICE
{ extensions CSTACommonArguments,
  noData NULL } See note [2]

PrivateEventInfo ::= NULL
-- The actual encoding of the private event is added here, replacing NULL
-- with another valid ASN.1 type
See note [3]

Note [1] See section { 4.4.8. } for details on the mapping of CSTACommonArguments.

4.3.7.2. Escape Services: Driver/Switch as 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.

NOTE: These services and events are not available to applications which are clients of drivers that register through the Cserver.

4.3.7.3. Maintenance Services

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

4.3.7.3.1. Device Status

This section describes the CSTA Maintenance Services that provide device status information. The device must be monitored by the application, with an active CSTA monitor association (that is, an active monitorCrossRefID), in order to receive this event. These events are unidirectional and always originate in the driver/switch domain.
### 4.3.7.3.2. System Status - Application as Client

This section defines the services that provide system level status information to the application from the switch/driver. The **TSAPI** system status services are implemented within the Cserver and are based on **System Status** requests sent by the switch/driver. In most cases there are no directly equivalent ECMA-180 services.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>typedef struct CSTABackInServiceEvent_t { DeviceID_t device; CSTAEventCause_t cause; } CSTABackInServiceEvent_t;</code></td>
<td><code>BackInServiceEventInfo ::= SEQUENCE { device DeviceID, cause EventCause OPTIONAL }</code></td>
<td><code>device &lt;- device [1]</code></td>
</tr>
<tr>
<td><code>typedef struct CSTAOutOfServiceEvent_t { DeviceID_t device; CSTAEventCause_t cause; } CSTAOutOfServiceEvent_t;</code></td>
<td><code>OutOfServiceEventInfo ::= SEQUENCE { device DeviceID, cause EventCause OPTIONAL }</code></td>
<td><code>device &lt;- device [1]</code></td>
</tr>
</tbody>
</table>

**Note [1]** See section [4.4.1.] for details on the mapping of DeviceID.

**Note [2]** See section [4.4.6.] for details on the mapping of EventCause.
typedef struct CSTASysStatEndedEvent_t
{
    Nulltype null;
} CSTASysStatEndedEvent_t;

Note [1] See section {4.4.5.} for details on the mapping of SystemStatus.

4.3.7.3.3. System Status - Driver/Switch as Client

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

NOTE: These services and events are not available to applications which are clients of drivers that register through the Cserver.

4.4. Mapping of CSTA-Defined Types

This section contains the details of mappings of various CSTA-defined data types. These are abstract syntax data types defined in ECMA-180 and generally have a counterpart C-Language-based definition in TSAPI. The direction of the mapping is indicated in the text accompanying each data type.

4.4.1. Device Identifiers

The DeviceID_t type defined in TSAPI is a NULL terminated string of up to 64 characters. ECMA-180, however, defines DeviceID as a choice of NumberDigits or DeviceNumber. NumberDigits is a character string and DeviceNumber is an integer. In the ECMA-180 to TSDI mapping direction, a DeviceID sent from the switch/driver encoded as NumberDigits is simply copied as a character string (that is, string of digits) into the applicable DeviceID_t type. Strings longer than 64 characters are silently truncated. A DeviceID encoded as DeviceNumber requires a special mapping. The DeviceID will be mapped by setting the first character of the DeviceID_t type to the character ‘\’ (that is, backslash) and setting the remaining characters to digits that equal the decimal representation of the integer value within the received DeviceID. For example, a DeviceID encoded as the DeviceNumber 12345 is mapped to the string "\12345". The reverse mapping applies in the opposite direction. A DeviceID_t type that contains the character ‘\’ as
the first character and digit characters in all remaining characters is encoded as a DeviceNumber. All other strings are encoded as NumberDigits.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type ↔ ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef enum DeviceIDType_t {</td>
<td></td>
<td>See explanation below.</td>
</tr>
<tr>
<td>DEVICE_IDENTIFIER = 0,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPLICIT_PUBLIC = 20,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PUBLIC_UNKNOWN = 30,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PUBLIC_INTERNATIONAL = 31,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PUBLIC_NATIONAL = 32,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PUBLIC_NETWORK_SPECIFIC = 33,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PUBLIC_SUBSCRIBER = 34,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PUBLIC_ABBREVIATED = 35,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPLICIT_PRIVATE = 40,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PRIVATE_UNKNOWN = 50,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PRIVATE_LEVEL3_REGIONAL_NUMBER = 51,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PRIVATE_LEVEL2_REGIONAL_NUMBER = 52,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PRIVATE_LEVEL1_REGIONAL_NUMBER = 53,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PRIVATE_PTN_SPECIFIC_NUMBER = 54,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PRIVATE_LOCAL_NUMBER = 55,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT_PRIVATE_ABBREVIATED = 56,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OTHER_PLAN = 60,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUNK_IDENTIFIER = 70,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUNK_GROUP_IDENTIFIER = 71</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} DeviceIDType_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef enum DeviceIDStatus_t {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID_PROVIDED = 0,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID_NOT_KNOWN = 1,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID_NOT_REQUIRED = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} DeviceIDStatus_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef struct ExtendedDeviceID_t {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeviceID_t deviceID;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeviceIDType_t deviceIDType;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeviceIDStatus_t deviceIDStatus;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} ExtendedDeviceID_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef ExtendedDeviceID_t CallingDeviceID_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef ExtendedDeviceID_t CalledDeviceID_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef ExtendedDeviceID_t SubjectDeviceID_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef ExtendedDeviceID_t RedirectionDeviceID_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>typedef ExtendedDeviceID_t OtherPlan;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OtherPlan ::= OCTET STRING</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| The CallingDeviceID, CalledDeviceID, SubjectDeviceID, and RedirectionDeviceID ECMA-180-defined types each have their counterpart in TSAPI in CallingDeviceID_t, CalledDeviceID_t, SubjectDeviceID_t, and RedirectionDeviceID_t, respectively. Although the ECMA-180 encoding and TSAPI representations are structured differently, they are essentially equivalent. These types are mapped only in the ECMA-180 to TSDI direction. Any IA5String or OCTET STRING (that is, for OtherPlan type) encoded in an ExtendedDeviceID is copied to the deviceID member of the ExtendedDeviceID_t structure. Strings longer than 64 characters are truncated. If the OtherPlan type encoding is chosen, the rules for mapping OCTET_STRING types described in section 4.5. apply. The deviceIDType member of ExtendedDeviceID_t is chosen to reflect the ECMA-180 type that was encoded. Note that the values of TRUNK_IDENTIFIER and TRUNK_GROUP_IDENTIFIER have no ECMA-180 equivalents and are therefore never used by this mapping. The value of the deviceIDStatus member of ExtendedDeviceID_t is chosen to reflect whether the CallingDeviceID, CalledDeviceID, SubjectDeviceID, or RedirectionDeviceID was encoded as an ExtendedDeviceID (deviceIDStatus = ID_PROVIDED), as notKnown (deviceIDStatus = ID_NOT_KNOWN), or as notRequired (deviceIDStatus = ID_NOT_REQUIRED).
For some CSTA service requests (that is, `cstaConsultationCall`, `cstaDivertCall`, `cstaMakeCall`, `cstaMakePredictiveCall` and `cstaRouteSelect`), the ECMA-180 operation argument contains a `CalledDeviceID` type parameter while the TSDI C-structure definition contains a `DeviceID_t` type parameter (not `CalledDeviceID_t` type). In such cases, the device identifier is always encoded as a `DeviceID` type (that is, select `ExtendedDeviceID` choice of `CalledDeviceID` type, and `DeviceID` choice of `ExtendedDeviceID` type). The `DeviceID` type mapping rules described above shall apply.

In some CSTA event reports (that is, `routeRequestEvent` and `routeUsedEvent`) the ECMA-180 operation argument contains `CalledDeviceID` and `CallingDeviceID` type parameters while the corresponding TSDI event C structure definitions contain `DeviceID_t` type members. In these cases, any `DeviceID`, `IA5String`, or `OCTET STRING` type encoded in the `CalledDeviceID` or `CallingDeviceID` type parameters will be mapped to the applicable `DeviceID_t` type member of the TSDI event structure. The information specifying the encoded type of `CalledDeviceID` or `CallingDeviceID` (that is, `deviceIdentifier`, `implicitPublic`, `explicitPublic`, `explicitPrivate`, or `other`) is not reflected in the TSDI event structure. If the `CalledDeviceID` or `CallingDeviceID` is encoded as `notKnown` or `notRequired`, or is omitted entirely, the corresponding TSDI event struct member is set to a NULL string.

### 4.4.2. Call and Connection Identifiers

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef enum ConnectionID_Device_t</td>
<td>ConnectionID ::= [APPLICATION 11] IMPLICIT SEQUENCE</td>
<td>callID &lt;-&gt; call</td>
</tr>
<tr>
<td>{</td>
<td></td>
<td>deviceID &lt;-&gt; device</td>
</tr>
<tr>
<td>STATIC_ID = 0,</td>
<td>[call [2] IMPLICIT OCTET STRING OPTIONAL,</td>
<td></td>
</tr>
<tr>
<td>DYNAMIC_ID = 1</td>
<td>device CHOICE</td>
<td></td>
</tr>
<tr>
<td>} ConnectionID_Device_t</td>
<td>{staticID DeviceID,</td>
<td></td>
</tr>
<tr>
<td>typedef struct ConnectionID_t</td>
<td>dynamicID[3] IMPLICIT OCTET STRING } OPTIONAL }</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long callID;</td>
<td>ConnectionID ::= [APPLICATION 11] IMPLICIT SEQUENCE</td>
<td></td>
</tr>
<tr>
<td>DeviceID_t deviceID;</td>
<td>[call [2] IMPLICIT OCTET STRING OPTIONAL,</td>
<td></td>
</tr>
<tr>
<td>ConnectionID_Device_t devIDType;</td>
<td>device CHOICE</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `ConnectionID_t` type defined in `TSAPI` is a structure containing a `callID` (defined as type `long`), a `deviceID` (defined as `DeviceID_t`), and a `devIDType` (defined as `ConnectionID_Device_t`). The ECMA-180 definition of `ConnectionID` contains a `call` and a `device`, and the `device` can be a `staticID` (that is, `DeviceID_t` type) or a `dynamicID` (OCTET STRING type). Connection identifiers are mapped in both directions.

The call identifier mapping (that is, `call` element to `callID` struct member, and vice-versa) can be controlled by the driver using the `mapping_descriptor` parameter of `cdiDriverRegister()`. Section {4.4.9.} describes the details of this mapping. If the `call` element of `ConnectionID` is omitted (that is, it is OPTIONAL), then the `callID` member of the `ConnectionID_t` struct is set to the value 0. The `call` element is encoded when mapping in the TSDI to ECMA-180 direction only if the supplied value of `callID` is not 0.

The device identifier mapping (that is, `device` element to `deviceID` and `devIDType` struct members, and vice-versa) is as follows. In the ECMA-180 to TSDI direction, a `device` encoded as a `staticID` (that is, `DeviceID_t` type) is mapped according to the `DeviceID` mapping specified in section {4.4.1.} and the `devIDType` struct member is set to the value `STATIC_ID`. A `device` encoded as a `dynamicID` (that is, `OCTET STRING` type) is mapped according to the `OCTET STRING` mapping rules described in section {4.5.} and the `devIDType` struct member is set to the value `DYNAMIC_ID`. If the `device` element of `ConnectionID` is omitted (that is, it is OPTIONAL), then the `deviceID` member of `ConnectionID_t` is set to a NULL string and the `devIDType` member is set to `STATIC_ID`. The reverse mapping applies in the opposite direction. The `device` element is included when mapping in the TSDI to ECMA-180 direction as long as `deviceID` is not a NULL string.

If the entire `ConnectionID` type element of a CSTA result or event is OPTIONAL, then it is mapped to the TSDI structure using a `callID` value of 0, a `deviceID` value of NULL string, and a `devIDType` value of `STATIC_ID`. In the TSDI to ECMA-180 direction, a `deviceID` value of NULL string and a `callID` value of 0 will cause the `ConnectionID` to be omitted if it is optional.
### 4.4.3. Connection States

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct Connection_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConnectionID_t party;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SubjectDeviceID_t staticDevice;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>} Connection_t;</td>
<td></td>
</tr>
<tr>
<td>typedef struct ConnectionList_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>int count;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Connection_t FAR *connection;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>} ConnectionList_t;</td>
<td></td>
</tr>
</tbody>
</table>

The `ConnectionList_t` type defined in TSAPI is a structure containing a `count` (defined as type `int`), and a `connection` pointer (defined as type `Connection_t FAR *`). The `connection` pointer points to an array of `Connection_t` structures, each containing a `party` (defined as type `ConnectionID_t`) and a `staticDevice` (defined as `SubjectDeviceID_t`). The `count` struct member reflects the number of elements in the array pointed to by `connection`. The ECMA-180 definition of `ConnectionList` is a choice of `connections` (defined as type `ConnectionIDList`) or `callInformation` (defined as type `CallInfo`). Connection lists are mapped in the ECMA-180 to TSDI direction only.

If `ConnectionList` is encoded as type `ConnectionIDList`, then the sequence of `ConnectionIDs` is mapped to an equivalent number of `Connection_t` structures with each `ConnectionID` in the sequence mapping to the `party` struct element of its corresponding `Connection_t` structure according to `ConnectionID` mapping rules specified in section 4.4.2. The `staticDevice` struct member will have `deviceID` set to a NULL string and `deviceIDStatus` set to `ID_NOT_KNOWN`.

If `ConnectionList` is encoded as type `CallInfo`, then the sequence specified by `CallInfo` is mapped to an equivalent number of `Connection_t` structures. For each member of the sequence, the `endpoint` (defined as type `ConnectionID`) is mapped to the `party` struct member of its corresponding `Connection_t` structure and the `staticEndpoint` (defined as type `DeviceID`) is mapped to the `staticDevice` struct member. If `staticEndpoint` is omitted (that is, it is OPTIONAL), then the `staticDevice` struct member will have `deviceID` set to a NULL string and `deviceIDStatus` set to `ID_NOT_KNOWN`. Otherwise, `deviceID` is mapped according to device identifier mapping rules described in section 4.4.1., `deviceIDType` is set to `DEVICE_IDENTIFIER`, and `devIDStatus` is set to `ID_PROVIDED`.

If `ConnectionList` is omitted (that is, it is optional), then the `count` member of `ConnectionList_t` is set to 0 and `connection` is set to NULL.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef enum LocalConnectionState_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS_NONE = -1,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS_NULL = 0,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS_INITIATE = 1,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS_ALERTING = 2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS_CONNECT = 3,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS_HOLD = 4,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS_QUEUED = 5,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS_FAIL = 6</td>
<td></td>
</tr>
<tr>
<td>} LocalConnectionState_t;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `LocalConnectionState_t` defined in TSAPI and `LocalConnectionState` defined in ECMA-180 are both `ENUMERATED` types with nearly equivalent sets of possible values. The only exception is the value `CS_NONE` which is never used by the mapping. The local connection state is mapped only in the ECMA-180 to TSDI direction.
4.4.4. Status Reporting

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
</table>
| No directly corresponding TSAPI C-Language Definition exists. The TSAPI cstaMonitorDevice, cstaMonitorCall, and cstaMonitorCallsViaDevice service requests are distinct services which are defined as a single MonitorStart service in ECMA-179. The MonitorType is used together with the MonitorObject information to select the type of monitoring desired. | MonitorType ::= ENUMERATED 
(call (0), 
device (1)) | See explanation below |
| No directly corresponding TSAPI C-Language Definition exists. The TSAPI cstaMonitorDevice, cstaMonitorCall, and cstaMonitorCallsViaDevice service requests are distinct services which are defined as a single MonitorStart service in ECMA-179. The MonitorObject is used together with the MonitorType information to select the type of monitoring desired. | MonitorObject ::= CSTAObject 
CSTAObject ::= CHOICE 
(device DeviceID, 
call ConnectionID) | See explanation below |

The ECMA-180 MonitorObject and MonitorType types are used to specify the type of desired monitoring as part of a Monitor Start service request. There are no equivalent TSAPI-defined types because TSAPI defines three distinct services (that is, cstaMonitorDevice, cstaMonitorCall, and cstaMonitorCallsViaDevice) in place of the single ECMA-179 defined MonitorStart. If cstaMonitorDevice is invoked, the monitorObject element of MonitorStartArgument will be set to device according to the DeviceID type mapping rules defined in section {4.4.1.}, and monitorType will be set to call. If cstaMonitorCall is invoked, the monitorObject element of MonitorStartArgument will be encoded as a call according to the ConnectionID type mapping rules defined in section {4.4.2.}, and monitorType will be set to call. If cstaMonitorCallsViaDevice is invoked, the monitorObject element of MonitorStartArgument will be encoded as a device according to the DeviceID type mapping rules defined in section {4.4.1.}, and monitorType will be set to call. This mapping only occurs in the TSDI to ECMA-180 direction.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef long MonitorCrossRefID_t;</td>
<td>MonitorCrossRefID ::= [APPLICATION 21] IMPLICIT OCTET STRING</td>
<td>See explanation below</td>
</tr>
</tbody>
</table>

The MonitorCrossRefID_t type is defined in TSAPI as a long, and the ECMA-180 equivalent type of MonitorCrossRefID is defined as type OCTET STRING. The monitor cross reference identifier is mapped in both the TSDI to ECMA-180 and ECMA-180 to TSDI directions. The mapping can be controlled by the driver using the mapping_descriptor parameter of cdDriverRegister(). Section {4.4.9.} describes the details of this mapping. The monitor cross reference identifier is mandatory in any ECMA-180 APDU that contains it.
The CSTAMonitorFilter_t type in TSAPI is defined as a structure containing the members call (defined as type CSTACallFilter_t), feature (defined as type CSTAFeatureFilter_t), agent (defined as type CSTAAgentFilter_t), maintenance (defined as type CSTAMaintenanceFilter_t), and private (defined as type BOOLEAN). These are mapped to the elements of the ECMA-180 defined type MonitorFilter, call (defined as type CallFilter), feature (defined as type FeatureFilter), agent (defined as type AgentFilter), maintenance (defined as type MaintenanceFilter), and private (defined as type BOOLEAN), respectively. The mappings of the call, feature, agent, and maintenance elements are straightforward in that the ECMA-180 BIT STRING definitions are equivalent to the manifest constants defined by TSAPI for each filter class. The private member of CSTAMonitorFilter_t, although declared as type long, is treated as a Boolean value and therefore can be directly mapped to/from the private element of MonitorFilter. MonitorFilter is mapped in both the TSDI to ECMA-180 and ECMA-180 to TSDI directions. In the TSDI to ECMA-180 direction, the MonitorFilter is always encoded regardless of value. In the ECMA-180 to TSDI direction, if the monitorFilter element of MonitorStartResult or filterList element of ChangeMonitorFilterResult is omitted (that is, they are OPTIONAL), the monitorFilter member of CSTAMonitorConfEvent_t or monitorFilter element of CSTAChangeMonitorFilterConfEvent_t, respectively, is set to the same value as was supplied by the application in the corresponding Monitor Start (that is, cstaMonitorDevice, cstaMonitorCall, cstaMonitorCallsViaDevice) or Change Monitor Filter (that is, cstaChangeMonitorFilter) service request.
The `CSTASnapshotDeviceData_t` type defined in TSAPI is a structure containing a `count` (defined as type `int`) and an `info` pointer (defined as type `CSTASnapshotDeviceResponseInfo_t FAR *`). The `info` pointer points to an array of `CSTASnapshotDeviceResponseInfo_t` structures, each containing a `callIdentifier` (defined as type `ConnectionID_t`) and a `localCallState` (defined as `CSTACallState_t`). The `count` struct member reflects the number of elements in the array pointed to by `info`. The ECMA-180 definition of `SnapshotDeviceData` is a sequence of `SnapshotDeviceResponseInfo` types, which contains a `callIdentifier` (defined as type `ConnectionID`) and a `localCallState` (defined as `CallState`). Each `callIdentifier` element in the sequence of `CSTASnapshotDeviceResponseInfo_t` types is mapped to the `callIdentifier` member of its corresponding `CSTASnapshotDeviceResponseInfo_t` structure according to `ConnectionID` mapping rules specified in section {4.4.2.}. Similarly, the `localCallState` element in the sequence of `CSTASnapshotDeviceResponseInfo_t` types is mapped to the `localCallState` member of its corresponding `CSTASnapshotDeviceResponseInfo_t` structure according to `CallInfo` mapping rules defined later in this section. `CSTASnapshotDeviceData` is only mapped in the ECMA-180 to TSDI direction.

The `CSTASnapshotCallData_t` type defined in TSAPI is a structure containing a `count` (defined as type `int`), and an `info` pointer (defined as type `CSTASnapshotCallResponseInfo_t FAR *`). The `info` pointer points to an array of `CSTASnapshotCallResponseInfo_t` structures, each containing a `deviceOnCall` (defined as type `SubjectDeviceID_t`), a `callIdentifier` (defined as type `ConnectionID_t`), and a `localConnectionState` (defined as `LocalConnectionState_t`). The `count` struct member reflects the number of elements in the array pointed to by `info`. The ECMA-180 definition of `SnapshotCallData` is a sequence of `SnapshotCallResponseInfo` types, which contains a `deviceOnCall` (defined as type `SubjectDeviceID`), a `callIdentifier` (defined as type `ConnectionID`), and a `localConnectionState` (defined as type `LocalConnectionState`). Each `deviceOnCall` element in the sequence of `CSTASnapshotCallResponseInfo_t` types is mapped to the `deviceOnCall` member of its corresponding `CSTASnapshotCallResponseInfo_t` structure according to `SubjectDeviceID` mapping rules specified in section {4.4.1.}. Each `callIdentifier` element in the sequence of `CSTASnapshotCallResponseInfo_t` types is mapped to the `callIdentifier` member of its corresponding `CSTASnapshotCallResponseInfo_t` structure according to `ConnectionID` mapping rules specified in section {4.4.2.}. Finally, the `localConnectionState` element in the sequence of `CSTASnapshotCallResponseInfo_t` types is mapped to the `localConnectionState` member of its corresponding `CSTASnapshotCallResponseInfo_t` structure according to...
**LocalConnectionState** mapping rules defined in section {4.4.3.}. **CSTASnapshotCallData** is only mapped in the ECMA-180 to TSDI direction.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct CSTACallState_t{</td>
<td>CallState::= CHOICE</td>
<td>See explanation below</td>
</tr>
<tr>
<td>int count;</td>
<td>(compound [0] IMPLICIT CompoundCallState,</td>
<td></td>
</tr>
<tr>
<td>LocalConnectionState_t FAR *state;</td>
<td>simple [1] IMPLICIT SimpleCallState,</td>
<td></td>
</tr>
<tr>
<td>} CSTACallState_t;</td>
<td>unknown [2] IMPLICIT NULL}</td>
<td></td>
</tr>
</tbody>
</table>

See explanation below

The **CSTACallState_t** type defined in TSAPI is a structure containing a **count** (defined as type **int**), and a **state** pointer (defined as type **LocalConnectionState_t FAR **). The **state** pointer points to an array of type **LocalConnectionState_t**. The **count** struct member reflects the number of elements in the array pointed to by **state**. The ECMA-180 definition of **CallState** is a choice of **compound** (defined as type **CompoundCallState**), **simple** (defined as type **SimpleCallState**), or **unknown** (defined as type **NULL**). Call states are mapped in the ECMA-180 to TSDI direction only.

If **CallState** is encoded as type **CompoundCallState**, then the sequence of **LocalConnectionState** is mapped to an equivalent number of **LocalConnectionState_t** array elements and mapped according to the **LocalConnectionState** mapping rules specified in section {4.4.3.}.

If **CallState** is encoded as type **SimpleCallState**, then the **count** member of **CSTACallState_t** is set to the value 2 and the **state** pointer is set to point to an array of **LocalConnectionState_t** with 2 elements. The first element contains the **LocalConnectionState_t** value represented in the low-order nibble (that is, low order 4 bits of byte) of the **SimpleCallState** value. The second element contains the **LocalConnectionState_t** value represented in the high-order nibble (that is, high order 4 bits of byte) of the **SimpleCallState** value. For example, if the **SimpleCallState** value is **callPending** (that is, '01' H null-initiate), then the first array element is set to **CS_INITIATE** and the second array element is set to **CS_NULL**.

If **CallState** is encoded as type **NULL** (that is, **unknown**), then the **count** member of **CSTACallState_t** is set to the value 0 and the **state** pointer is set to NULL.

### 4.4.5. Device and Feature Types and Other Parameters

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef char AccountInfo_t[32];</td>
<td>AccountInfo ::= OCTET STRING</td>
<td>See explanation below</td>
</tr>
<tr>
<td>typedef char AuthCode_t[32];</td>
<td>AuthCode ::= OCTET STRING</td>
<td>See explanation below</td>
</tr>
<tr>
<td>typedef char AgentID_t[32];</td>
<td>AgentID ::= OCTET STRING</td>
<td>See explanation below</td>
</tr>
</tbody>
</table>
The `AccountInfo_t`, `AuthCode_t`, `AgentID_t`, and `AgentPassword_t` types are defined in TSAPI as character strings of maximum length 32, and the ECMA-180 equivalent types of `AccountInfo`, `AuthCode`, `AgentID`, and `AgentPassword`, respectively, are defined as type `OCTET STRING`. These types are all mapped according to the `OCTET STRING` mapping rules described in section {4.5.}. `OCTET STRING`s longer than 32 are silently truncated. The first two types are mapped in the ECMA-180 to TSDI direction only and the last two types are mapped in both directions. When mapping in the ECMA-180 to TSDI direction, if an element defined by one of these types is omitted (that is, when the element is optional), then the corresponding member of the TSDI structure is set to a NULL string. In the opposite direction, if the element defined as type `AgentID` or `AgentPassword` is optional, it will be omitted if the corresponding TSDI struct member is a NULL string.

The `AgentGroup_t` type is defined in TSAPI as a `DeviceID_t`, and the ECMA-180 equivalent type `AgentGroup` is defined as a `DeviceID`. This type is mapped according to `DeviceID` mapping rules specified in section {4.4.1.}. The Agent Group type is mapped in both the ECMA-180 to TSDI and TSDI to ECMA-180 directions.

The `AgentMode_t` type is defined in TSAPI as an `enum` type that is used in the `cstaSetAgentState` service request and does not have a direct ECMA-180 equivalent. ECMA-180, however, defines an `AgentParameter` type that defines the same agent mode values as `AgentMode_t` but also contains embedded arguments for those modes. The `loggedIn` and `loggedOut` elements of `AgentParameter` are defined as types `LoggedOnInfo` and `LoggedOffInfo`, respectively, and contain agent login and logout parameters. The equivalent TSAPI parameters are included in the definition of the `CSTASetAgentState_t` service request structure. For the agent login case, the `agentID`, `agentPassword`, and `agentGroup` elements of `LoggedOnInfo` are mapped from the equivalently named structure members of `CSTASetAgentState_t`. For the agent logout case, the `agentID` and `agentGroup` elements of `LoggedOffInfo` are mapped from the equivalently named structure members of `CSTASetAgentState_t`. The mappings involving `AgentParameter` are only in the TSDI to ECMA-180 direction. The values of the `agentID`, `agentPassword`, and/or `agentGroup` members of `CSTASetAgentState_t` are ignored if they do not apply to the selected agent mode.
The `AgentState_t` type is defined in TSAPI as an `enum` type, and the ECMA-180 equivalent type of `AgentState` is defined as type `ENUMERATED`. The possible values of `AgentState_t` and `AgentState` are identical and map in a straightforward manner as shown in the table above. The mapping only occurs in the ECMA-180 to TSDI direction.

The `AllocationState_t` type is defined in TSAPI as an `enum` type, and the ECMA-180 equivalent type of `AllocationState` is defined as type `ENUMERATED`. The possible values of `AllocationState_t` and `AllocationState` are identical and map in a straightforward manner as shown in the table above. The `AllocationState` type is always encoded even if the value matches a default value. The mapping only occurs in the TSDI to ECMA-180 direction.

The ECMA-180 defined type `ConnectionDetails` is used to define arguments to CSTA service requests that typically operate on two calls or connections (that is, `cstaAlternateCall`, `cstaConferenceCall`, `cstaReconnectCall`, and `cstaTransferCall` service requests). The type is defined as a choice of `heldCall` (defined as type `ConnectionID`), `activeCall` (defined as type `ConnectionID`), or `bothCalls` (defined as sequence of `heldCall` and `activeCall` which are both `ConnectionID`). The definition allows for either the held call, active call, or both held and active calls to be specified. The information contained in `ConnectionDetails` is mapped from the individual service request structures for the previously mentioned services, each of which contains two `ConnectionID` type members. The choice of `heldCall`, `activeCall`, or `bothCalls` encoding depends on the values of the these two `ConnectionID` type members. If the value is `callID` equals 0 and `deviceID` equals NULL string, then the corresponding `ConnectionID` is not encoded. If neither or both `ConnectionID` have this value, then `bothCalls` is encoded. This mapping only occurs in the TSDI to ECMA-180 direction.

The ECMA-180 defined type `DeviceInfo` is used to define the return results of the query device information service request. It is defined as a sequence of `deviceID` (defined as type `DeviceID`), `deviceType` (defined as type `DeviceType`) and `deviceClass` (defined as type `DeviceClass`). The elements of `DeviceInfo` are mapped into the same named members.
of the TSAPI `CSTAQueryDeviceInfoConfEvent_t` structure as shown in the table above. The `deviceID` element is mapped according to `DeviceID` mapping rules described in section 4.4.1.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef enum DeviceType_t</td>
<td>DeviceType := ENUMERATED</td>
<td>Value mapping:</td>
</tr>
<tr>
<td>{ DT_STATION = 0,</td>
<td>(station (0),</td>
<td>DT_STATION &lt;- station</td>
</tr>
<tr>
<td>DT_LINE = 1,</td>
<td>line (1),</td>
<td>DT_LINE &lt;- line</td>
</tr>
<tr>
<td>DT_BUTTON = 2,</td>
<td>button (2),</td>
<td>DT_BUTTON &lt;- button</td>
</tr>
<tr>
<td>DT_ACD = 3,</td>
<td>aCD (3),</td>
<td>DT_ACD &lt;- aCD</td>
</tr>
<tr>
<td>DT_TRUNK = 4,</td>
<td>trunk (4),</td>
<td>DT_TRUNK &lt;- trunk</td>
</tr>
<tr>
<td>DT_OPERATOR = 5,</td>
<td>operator (5),</td>
<td>DT_OPERATOR &lt;- operator</td>
</tr>
<tr>
<td>DT_STATION_GROUP = 16,</td>
<td>station-group (16),</td>
<td>DT_STATION_GROUP &lt;- station-group</td>
</tr>
<tr>
<td>DT_LINE_GROUP = 17,</td>
<td>line-group (17),</td>
<td>DT_LINE_GROUP &lt;- line-group</td>
</tr>
<tr>
<td>DT_BUTTON_GROUP = 18,</td>
<td>button-group (18),</td>
<td>DT_BUTTON_GROUP &lt;- button-group</td>
</tr>
<tr>
<td>DT_ACD_GROUP = 19,</td>
<td>aCD-group (19),</td>
<td>DT_ACD_GROUP &lt;- aCD-group</td>
</tr>
<tr>
<td>DT_TRUNK_GROUP = 20,</td>
<td>trunk-group (20),</td>
<td>DT_TRUNK_GROUP &lt;- trunk-group</td>
</tr>
<tr>
<td>DT_OPERATOR_GROUP = 21,</td>
<td>operator-group (21),</td>
<td>DT_OPERATOR_GROUP &lt;- operator-group</td>
</tr>
<tr>
<td>DT_OTHER = 255</td>
<td>other (255)</td>
<td>DT_OTHER &lt;- other</td>
</tr>
</tbody>
</table>

The `DeviceType_t` type is defined in TSAPI as an `enum` type, and the ECMA-180 equivalent type of `DeviceType` is defined as type `ENUMERATED`. The possible values of `DeviceType_t` and `DeviceType` are identical and map in a straightforward manner as shown in the table above. If a `DeviceType` element is omitted (that is, it is OPTIONAL), the `DeviceType_t` type field is set to `DT_STATION`. The mapping only occurs in the ECMA-180 to TSDI direction.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef unsigned char DeviceClass_t;</td>
<td>DeviceClass := BIT STRING</td>
<td>Value mapping:</td>
</tr>
<tr>
<td>#define DC_VOICE 0x80</td>
<td>(voice (0),</td>
<td>DC_VOICE &lt;- voice</td>
</tr>
<tr>
<td>#define DC_DATA 0x40</td>
<td>data (1),</td>
<td>DC_DATA &lt;- data</td>
</tr>
<tr>
<td>#define DC_IMAGE 0x20</td>
<td>image (2),</td>
<td>DC_IMAGE &lt;- image</td>
</tr>
<tr>
<td>#define DC_OTHER 0x10</td>
<td>other (3),</td>
<td>DC_OTHER &lt;- other</td>
</tr>
</tbody>
</table>

The `DeviceClass_t` type in TSAPI is defined as an `unsigned char` for which manifest constants are defined to specify the meanings of individual bits within `DeviceClass_t`. ECMA-180 defines an equivalent `DeviceClass` type which is defined as a `BIT STRING`. The `BIT STRING` definition is equivalent to the manifest constants defined by TSAPI. If a `DeviceClass` type element is omitted (that is, it is OPTIONAL), the `DeviceClass_t` type field is set to 0. This type is mapped in the ECMA-180 to TSDI direction only.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No directly corresponding TSAPI C-Structure Definition exists. The TSAPI cstaDeflectCall, cstaPickupCall, and cstaGroupPickupCall service request structures whose fields are mapped into DivertInfo.</td>
<td>DivertInfo := CHOICE</td>
<td>See explanation below</td>
</tr>
<tr>
<td>(deflect [0] IMPLICIT SEQUENCE</td>
<td>(callToBeDiverted ConnectionID newDestination CalledDeviceID )</td>
<td></td>
</tr>
<tr>
<td>pickup [1] IMPLICIT SEQUENCE</td>
<td>(callToBePickedUp ConnectionID requestingDevice DeviceID )</td>
<td></td>
</tr>
<tr>
<td>group [2] DeviceID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The ECMA-180 `DivertInfo` type is used to define the parameters for the `Divert Call` service. There is no TSAPI equivalent because TSAPI defines three distinct services (cstaDeflectCall(), cstaPickupCall(), and cstaGroupPickupCall()) in place of the single ECMA-179 service. Each choice within `DivertInfo` (that is, `deflect`, `pickup`, and `group`), however, corresponds to one of the TSAPI services. For the `deflect` choice, the sequence elements `callToBeDiverted` and `newDestination` are mapped from the `deflectCall` and `calledDevice` members of `CSTADeflectCall_t`, respectively. For the `pickup` choice, the sequence elements `callToBePickedUp` and `requestingDevice` are mapped from the `deflectCall` and `calledDevice` members of `CSTAPickupCall_t`, respectively.
Finally, for the group choice, the device identifier is mapped from the pickupDevice member of CSTAGroupPickupCall_t. Note that the deflectCall member of CSTAGroupPickupCall_t is ignored for this mapping. The callToBeDiverted and callToBePickedUp elements are mapped according to ConnectionID mapping rules described in section 4.4.2 and the newDestination, requestingDevice, and group elements are mapped according to DeviceID mapping rules described in section 4.4.1. DivertInfo is mapped in the TSDI to ECMA-180 direction only.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef enum Feature_t</td>
<td>FeatureInfo ::= CHOICE</td>
<td>See explanation below</td>
</tr>
<tr>
<td>{ FT_CAMP_ON = 0,</td>
<td>[campon] IMPLICIT ConnectionID,</td>
<td></td>
</tr>
<tr>
<td>FT_CALL_BACK = 1,</td>
<td>[callback] IMPLICIT ConnectionID,</td>
<td></td>
</tr>
<tr>
<td>FT_INTRUDE = 2</td>
<td>[intrude] IMPLICIT ConnectionID }</td>
<td></td>
</tr>
</tbody>
</table>

The Feature_t type in TSAPI is defined as an enum type, and the associated ECMA-180 type of FeatureInfo is defined as a choice of campon, callback, or intrude (all defined as type ConnectionID). FeatureInfo is used by the cstaCallCompletion service and includes the feature type as well as the connection identifier argument. Feature_t is meant to indicate the feature only (that is, the connection identifier is a separate member of CSTACallCompletion_t). Feature_t together with the connection identifier (that is, call member) from CSTACallCompletion_t are mapped to the corresponding element of Feature and the according to ConnectionID mapping rules described in section 4.4.2. This mapping occurs in the TSDI to ECMA-180 direction only.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct ListForwardParameters_t</td>
<td>ListForwardParameters ::= SEQUENCE OF</td>
<td>See explanation below</td>
</tr>
<tr>
<td>{ short count;</td>
<td>SEQUENCE</td>
<td></td>
</tr>
<tr>
<td>ForwardInfo_info[7];</td>
<td>{forwardingType ForwardingType,</td>
<td></td>
</tr>
<tr>
<td>} ListForwardParameters_t;</td>
<td>forwardDN NumberDigits }</td>
<td></td>
</tr>
</tbody>
</table>

The ListForwardParameters_t type defined in TSAPI is a structure containing a count (defined as type short), and a param array of size 7 (defined as type ForwardingInfo_t). The count member reflects the actual number of elements in the array param. The ForwardingInfo_t is a structure containing a forwardingType (defined as type ForwardingType_t), forwardingOn defined as type Boolean, and forwardDN defined as type DeviceID. The ForwardingType_t type is defined as an enum type with 7 possible values. The number of possible values is directly related to the maximum size of the param array within ListForwardParameters_t. ECMA-180 defines a type equivalent to ListForwardParameters_t called ListForwardParameters. The information is structured differently, however. ListForwardParameter is a sequence of sequence including forwardingType (defined as type ForwardingType) and forwardDN (defined as type NumberDigits). ForwardingType is defined as type
ENUMERATED with 14 possible values. The values are identical to those of `ForwardingType_t` except that there is an ON and OFF value for each possible value of `ForwardingType_t`. This forwarding ON/OFF indication is equivalently specified by the `forwardingOn` (defined as type `BOOLEAN`) member of `ForwardingInfo_t`. ListForwardParameters is mapped only in the ECMA-180 to TSDI directions. The `count` member of `ListForwardParameters_t` is set to reflect the number of members in the `ListForwardParameters` sequence. The `forwardingType` element is mapped into the `forwardingType` and `forwardingOn` members of `ForwardingInfo_t` by setting `forwardingType` to indicate the forwarding type and setting the `forwardingOn` Boolean to TRUE for a forwarding ON setting and FALSE for a forwarding OFF setting. The `forwardDN` element is mapped to the same named member of `ForwardingInfo_t` using the `DeviceID` mapping rules described in section {4.4.1.}, except that `forwardDN` is always assumed to be `NumberDigits` (not `DeviceNumber`). A reverse mapping of `ForwardingInfo_t` to the ECMA-180 type `ForwardParameter` is also performed. The `ForwardParameter` is identical to the sequence within `ListForwardParameters` except that `forwardDN` is optional. If `forwardDN` in `ForwardInfo_t` is a NULL string then the `forwardDN` element will be omitted from `ForwardParameter`, otherwise, it will be included. `forwardDN` is always assumed to be of type `NumberDigits`. The `forwardType` and `forwardOn` members are mapped in an inverse manner to that described above.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No directly corresponding TSAPI C-Language Type Definition exists. The TSAPI CSTAQueuedEvent event structure uses a short type for the information contained in NoOfCallsInQueue.</td>
<td>NoOfCallsInQueue := INTEGER</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>

The ECMA-180 `NoOfCallsInQueue` type (defined as type `INTEGER`) has no TSAPI equivalent. TSAPI defines the `numberQueued` member of the `CSTAQueuedEvent_t` event structure as type `short`. This type is mapped in the ECMA-180 to TSDI direction only. If the `numberQueued` element of `QueueEventInfo` is omitted (that is, it is OPTIONAL), the `numberQueued` member of `CSTAQueuedEvent_t` is set to -1.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No directly corresponding TSAPI C-Language Type Definition exists. The TSAPI <code>cstaQueryMwi</code>, <code>cstaQueryDnd</code>, <code>cstaQueryLastNumber</code>, <code>cstaQueryDeviceInfo</code>, and <code>cstaQueryAgentState</code> services are distinct services, whereas ECMA-179 combines these 6 services into a single <code>QueryDevice</code> service and uses <code>QueryDeviceFeature</code> to indicate the query type.</td>
<td>QueryDeviceFeature ::= ENUMERATED {msgWaitingOn (0), doNotDisturbOn (1), forward (2), lastDialedNumber (3), deviceInfo (4), agentState (5)}</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>

The ECMA-180 `QueryDeviceFeature` type (defined as type `ENUMERATED`) is used to specify the query type in a `Query Device` service request. TSAPI defines 6 distinct services (that is, `cstaQueryMwi`, `cstaQueryDnd`, `cstaQueryLastNumber`, `cstaQueryDeviceInfo`, and `cstaQueryAgentState`) for this single ECMA-180 service. The feature element of `QueryDeviceArgument` is set according to which TSAPI service has been invoked.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No directly corresponding TSAPI C-Language Type Definition exists. The TSAPI <code>cstaQueryMwi</code>, <code>cstaQueryDnd</code>, <code>cstaQueryLastNumber</code>, <code>cstaQueryDeviceInfo</code>, and <code>cstaQueryAgentState</code> services are distinct services, whereas ECMA-179 combines these 6 services into a single <code>QueryDevice</code> service and uses <code>QueryDeviceFeature</code> to report query results. TSAPI defines distinct confirmation event structures for each of these services.</td>
<td>QueryDeviceInformation ::= CHOICE {msgWaitingOn [0] IMPLICIT BOOLEAN, doNotDisturbOn [1] IMPLICIT BOOLEAN, forward [2] IMPLICIT ListForwardParameters, lastDialed [3] IMPLICIT <code>NumberDigits</code>, deviceInfo [4] IMPLICIT <code>DeviceInfo</code>, agentState [5] IMPLICIT <code>AgentState</code> }</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>

The ECMA-180 `QueryDeviceInformation` type, defined as a `CHOICE` of `msgWaitingOn` (defined as type `BOOLEAN`), `doNotDisturbOn` (defined as type `BOOLEAN`), `forward` (defined as type `ListForwardParameters`),

The ECMA-180 NoOfCallsInQueue type (defined as type INTEGER) has no TSAPI equivalent. TSAPI defines the numberQueued member of the `CSTAQueuedEvent_t` event structure as type short. This type is mapped in the ECMA-180 to TSDI direction only. If the numberQueued element of `QueueEventInfo` is omitted (that is, it is OPTIONAL), the numberQueued member of `CSTAQueuedEvent_t` is set to -1.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No directly corresponding TSAPI C-Language Type Definition exists. The TSAPI <code>cstaQueryMwi</code>, <code>cstaQueryDnd</code>, <code>cstaQueryLastNumber</code>, <code>cstaQueryDeviceInfo</code>, and <code>cstaQueryAgentState</code> services are distinct services, whereas ECMA-179 combines these 6 services into a single <code>QueryDevice</code> service and uses <code>QueryDeviceFeature</code> to indicate the query type.</td>
<td>QueryDeviceFeature ::= ENUMERATED {msgWaitingOn (0), doNotDisturbOn (1), forward (2), lastDialedNumber (3), deviceInfo (4), agentState (5) }</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>
The ECMA-180 `SetDeviceFeature` type, defined as a CHOICE of `msgWaitingOn` (defined as type `BOOLEAN`), `doNotDisturbOn` (defined as type `BOOLEAN`), `forward` (defined as type `ForwardParameter`), and `aRequestedAgentState` (defined as type `AgentParameter`) is used to specify parameters for the `Set Feature` service request. TSAPI defines 4 distinct services (that is, `cstaSetyMwi`, `cstaSetDnd`, and `cstaSetAgentState`) for this single ECMA-180 service. The `SetDeviceFeature` information is contained in the individual service request structures `CSTASetMwi_t`, `CSTASetDnd_t`, `CSTASetFwd_t`, and `CSTASetAgentState_t`, respectively. The mapping occurs only in the TSDI to ECMA-180 direction.

The `SystemStatus_t` type is defined in TSAPI as an `enum` type, and the ECMA-180 equivalent type of `SystemStatus` is defined as type `ENUMERATED`. The possible values of `SystemStatus_t` and `SystemStatus` are identical and map in a straightforward manner as shown in the table above. The mapping occurs both in the ECMA-180 to TSDI and TSDI to ECMA-180 directions.

The `SelectValue_t` type is defined in TSAPI as an `enum` type, and the ECMA-180 equivalent type of `SelectValue` is defined as type `ENUMERATED`. The possible values of `SelectValue_t` and `SelectValue` are identical and map in a straightforward manner as shown in the table above. The mapping only occurs in the ECMA-180 to TSDI direction. If a `SelectValue` element is omitted (that is, it is OPTIONAL), the corresponding `SelectValue_t` type member will be set to `SV_NORMAL`.
The ECMA-180 `ReserveConnection` type (defined as type `BOOLEAN`) has no TSAPI equivalent. TSAPI defines the `reservation` member of the `CSTAHoldCall_t` structure as type `Boolean`. This type is mapped in the TSDI to ECMA-180 direction only.

The ECMA-180 `PriorityValue` type (defined as type `BOOLEAN`) has no TSAPI equivalent. TSAPI defines the `priority` member of the `CSTARouteRequestEvent_t` structure as type `Boolean`. This type is mapped in the ECMA-180 to TSDI direction only. If `PriorityValue` is omitted (that is, it is OPTIONAL), the `priority` member of `CSTARouteRequestEvent_t` is set to `FALSE`.

The `RetryValue_t` type is defined in TSAPI as type `short`, and the equivalent ECMA-180 type `RetryValue` is defined as a choice of `noListAvailable` (defined as type `BOOLEAN`), `noCountAvailable` (defined as type `BOOLEAN`), and `retryCount` (defined as type `INTEGER`). This type is mapped in the TSDI to ECMA-180 direction only. The manifest constants `noListAvailable` and `noCountAvailable` are used to select the same named choices of `RetryValue`. These two elements are always encoded with a `BOOLEAN` value of `true`. Any other value of the `RetryValue_t` type is encoded as choice `retryCount`. `RetryValue` is always encoded even when it is optional.

The `SetUpValues_t` type is defined in TSAPI as a structure containing a `length` (defined as type `int`) and `value` (defined as `char FAR *`), and the equivalent ECMA-180 type `SetUpValues` is defined as type `OCTET STRING`. This type is mapped in both the ECMA-180 to TSDI and TSDI to ECMA-180 directions. In the ECMA-180 to TSDI direction, `length` is set to reflect the `OCTET STRING` length and `value` is set to point to the contents of the `OCTET STRING`. If `SetUpValues` is omitted (that is, it is OPTIONAL), then `length` is set to 0 and `value` is set to NULL. In the TSDI to ECMA-180 direction `length` defines the length `OCTET STRING` to encode whose contents are pointed to by `value`. If `value` is NULL or `length` is 0, `SetUpValues` is not encoded.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReserveConnection ::= BOOLEAN</td>
<td>ReserveConnection ::= BOOLEAN</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityValue ::= BOOLEAN</td>
<td>PriorityValue ::= BOOLEAN</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef short RetryValue_t;</td>
<td>RetryValue ::= CHOICE</td>
<td>Value mapping:</td>
</tr>
<tr>
<td>#define noListAvailable -1</td>
<td>(noListAvailable [0] IMPLICIT BOOLEAN,</td>
<td>noListAvailable -&gt; noListAvailable</td>
</tr>
<tr>
<td>#define noCountAvailable -2</td>
<td>noCountAvailable [1] IMPLICIT BOOLEAN,</td>
<td>noCountAvailable -&gt; noCountAvailable</td>
</tr>
<tr>
<td></td>
<td>retryCount [2] IMPLICIT INTEGER )</td>
<td>(other value) -&gt; retryCount</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef struct SetUpValues_t</td>
<td>SetUpValues ::= OCTET STRING</td>
<td>See explanation below.</td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int length;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned char FAR * value;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} SetUpValues_t</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RouteUsedFlag ::= BOOLEAN</td>
<td>RouteUsedFlag ::= BOOLEAN</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>
The ECMA-180 RouteUsedFlag type (defined as type BOOLEAN) has no TSAPI equivalent. TSAPI defines the routeUsedReq member of the CSTARouteSelectRequest_t structure as type Boolean. This type is mapped in the TSDI to ECMA-180 direction only. The routeUsedReq element of RouteSelectRequestArgument is always encoded.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainValue := BOOLEAN</td>
<td>DomainValue ::= BOOLEAN</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>

The ECMA-180 DomainValue type (defined as type BOOLEAN) has no TSAPI equivalent. TSAPI defines the domain member of the CSTARouteUsedEvent_t event structure as type Boolean. This type is mapped in the ECMA-180 to TSDI direction only. If DomainValue is omitted (that is, it is OPTIONAL), the domain member of CSTARouteUsedEvent_t is set to FALSE.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef long RoutingCrossRefID_t;</td>
<td>RoutingCrossRefID ::= [APPLICATION 24] IMPLICIT OCTET STRING</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>

The RoutingCrossRefID_t type is defined in TSAPI as a long, and the ECMA-180 equivalent type of RoutingCrossRefID is defined as type OCTET STRING. The routing cross reference identifier is mapped in both the TSDI to ECMA-180 and ECMA-180 to TSDI directions. The mapping can be controlled by the driver using the mapping_deserializer parameter of cdIDriverRegister(). Section [4.4.9.] describes the details of this mapping. The routing cross reference identifier is mandatory in any ECMA-180 APDU that contains it.

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef long RouteRegisterReqID_t;</td>
<td>No equivalent ECMA-180 definition exists.</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>

The RouteRegisterReqID_t type is defined in TSAPI as a long, and has no ECMA-180 equivalent. This identifier is assigned by the Cserver and is never seen by the PBX driver. The RouteRegisterReqID_t is used to identify the registration of a client to become the routing server for a particular routing device, or to become the default routing server.
### Universal Failure and Event Cause

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef enum CSTAUniversalFailure_t {</td>
<td>UniversalFailure ::= CHOICE</td>
<td>See explanation below.</td>
</tr>
<tr>
<td>GENERIC_UNSPECIFIED = 0,</td>
<td>[1] IMPLICIT Operations,</td>
<td></td>
</tr>
<tr>
<td>GENERIC_OPERATION = 1,</td>
<td>[2] IMPLICIT StateIncompatibility,</td>
<td></td>
</tr>
<tr>
<td>REQUEST_INCOMPATIBLE_WITH_OBJECT = 2,</td>
<td>[3] IMPLICIT SystemResourceAvailability,</td>
<td></td>
</tr>
<tr>
<td>VALUE_OUT_OF_RANGE = 3,</td>
<td>[4] IMPLICIT SubscribedResourceAvailability,</td>
<td></td>
</tr>
<tr>
<td>OBJECT_NOT_KNOWN = 4,</td>
<td>[5] IMPLICIT PerformanceManagement,</td>
<td></td>
</tr>
<tr>
<td>INVALID_CALLING_DEVICE = 5,</td>
<td>[6] IMPLICIT SecurityError,</td>
<td></td>
</tr>
<tr>
<td>INVALID_CALLED_DEVICE = 6,</td>
<td>[7] IMPLICIT NULL,</td>
<td></td>
</tr>
<tr>
<td>INVALID_FORWARDING_DESTINATION = 7,</td>
<td>CSTATPrivateData</td>
<td></td>
</tr>
<tr>
<td>PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE = 8,</td>
<td>Operations ::= ENUMERATED</td>
<td></td>
</tr>
<tr>
<td>PRIVILEGE_VIOLATION_ON_CALLED_DEVICE = 9,</td>
<td>{generic (1),</td>
<td></td>
</tr>
<tr>
<td>PRIVILEGE_VIOLATION_ON_CALLINGDEVICE = 10,</td>
<td>requestIncompatibleWithObject (2),</td>
<td></td>
</tr>
<tr>
<td>INVALID_CSTA_CALL_IDENTIFIER = 11,</td>
<td>valueOutOfRange (3),</td>
<td></td>
</tr>
<tr>
<td>INVALID_CSTADEVICE_IDENTIFIER = 12,</td>
<td>objectNotKnown (4),</td>
<td></td>
</tr>
<tr>
<td>INVALID_CSTA_CONNECTION_IDENTIFIER = 13,</td>
<td>invalidCallingDevice (5),</td>
<td></td>
</tr>
<tr>
<td>INVALID_DESTINATION = 14,</td>
<td>invalidCalledDevice (6),</td>
<td></td>
</tr>
<tr>
<td>INVALID_FEATURE = 15,</td>
<td>invalidForwardingDestination (7),</td>
<td></td>
</tr>
<tr>
<td>INVALID_ALLOCATION_STATE = 16,</td>
<td>privilegeViolationOnSpecifiedDevice (8),</td>
<td></td>
</tr>
<tr>
<td>INVALID_CROSS_REF_ID = 17,</td>
<td>privilegeViolationOnCalledDevice (9),</td>
<td></td>
</tr>
<tr>
<td>INVALID_OBJECT_TYPE = 18,</td>
<td>privilegeViolationOnCallingDevice (10),</td>
<td></td>
</tr>
<tr>
<td>SECURITY_VIOLATION = 19,</td>
<td>invalidCSTAAcceptIdentifier (11),</td>
<td></td>
</tr>
<tr>
<td>GENERIC_STATE_INCOMPATIBILITY = 21,</td>
<td>invalidCSTADeviceIdentifier (12),</td>
<td></td>
</tr>
<tr>
<td>INVALID_OBJECT_STATE = 22,</td>
<td>invalidCSTAConnectionIdentifier (13),</td>
<td></td>
</tr>
<tr>
<td>INVALID_CONNECTION_ID_FOR_ACTIVE_CALL = 23,</td>
<td>invalidDestination (14),</td>
<td></td>
</tr>
<tr>
<td>NO_ACTIVE_CALL = 24,</td>
<td>invalidFeature (15),</td>
<td></td>
</tr>
<tr>
<td>NO_HELD_CALL = 25,</td>
<td>invalidAllocationState (16),</td>
<td></td>
</tr>
<tr>
<td>NO_CALL_TO_CLEAR = 26,</td>
<td>invalidCrossRefID (17),</td>
<td></td>
</tr>
<tr>
<td>NO_CONNECTION_TO_CLEAR = 27,</td>
<td>invalidObjectType (18),</td>
<td></td>
</tr>
<tr>
<td>NO_CALL_TO_ANSWER = 28,</td>
<td>securityViolation (19)</td>
<td></td>
</tr>
<tr>
<td>NO_CALL_TO_COMPLETE = 29,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GENERIC_SYSTEM_RESOURCE_AVAILABILITY = 31,</td>
<td>StateIncompatibility ::= ENUMERATED</td>
<td></td>
</tr>
<tr>
<td>SERVICE_BUSY = 32,</td>
<td>(generic (1),</td>
<td></td>
</tr>
<tr>
<td>RESOURCE_BUSY = 33,</td>
<td>invalidObjectState (2),</td>
<td></td>
</tr>
<tr>
<td>RESOURCE_OUT_OF_SERVICE = 34,</td>
<td>noActiveCall (3),</td>
<td></td>
</tr>
<tr>
<td>NETWORK_BUSY = 35,</td>
<td>noHeldCall (4),</td>
<td></td>
</tr>
<tr>
<td>NETWORK_OUT_OF_SERVICE = 36,</td>
<td>noCallToClear (5),</td>
<td></td>
</tr>
<tr>
<td>OVERALL_MONITOR_LIMIT_EXCEEDED = 37,</td>
<td>noConnectionToClear (6),</td>
<td></td>
</tr>
<tr>
<td>CONFERENCE_MEMBER_LIMIT_EXCEEDED = 38,</td>
<td>noCallToAnswer (7),</td>
<td></td>
</tr>
<tr>
<td>GENERIC_SUBSCRIBED_COUNTRY_AVAILABILITY = 41,</td>
<td>noCallToComplete (8),</td>
<td></td>
</tr>
<tr>
<td>OBJECT_MONITOR_LIMIT_EXCEEDED = 42,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTERNAL_TRUNK_LIMIT_EXCEEDED = 43,</td>
<td>SubscribedResourceAvailability ::= ENUMERATED</td>
<td></td>
</tr>
<tr>
<td>OUTSTANDING_REQUEST_LIMIT_EXCEEDED = 44,</td>
<td>(generic (1),</td>
<td></td>
</tr>
<tr>
<td>GENERIC_PERFORMANCE_MANAGEMENT = 51,</td>
<td>objectMonitorLimitExceeded (2),</td>
<td></td>
</tr>
<tr>
<td>PERFORMANCE_LIMIT_EXCEEDED = 52,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(continued on next page)</td>
<td></td>
</tr>
</tbody>
</table>
UNSPECIFIED_SECURITY_ERROR = 60,
SEQUENCE_NUMBER_VIOLATED = 61,
TIME_STAMP_VIOLATED = 62,
PAC_VIOLATED = 63,
SEAL_VIOLATED = 64,
GENERIC_UNSPECIFIED_REJECTION = 70,
GENERIC_OPERATION_REJECTION = 71,
DUPLICATE_INVOCATION_REJECTION = 72,
UNRECOGNIZED_OPERATION_REJECTION = 73,
MISTYPED_ARGUMENT_REJECTION = 74,
RESOURCE_LIMITATION_REJECTION = 75,
ACS_HANDLE_TERMINATION_REJECTION = 76,
SERVICE_TERMINATION_REJECTION = 77,
REQUEST_TIMEOUT_REJECTION = 78,
REQUESTS_ON_DEVICE_EXCEEDED_REJECTION = 79,
UNRECOGNIZED_APDU_REJECTION = 80,
MISTYPED_APDU_REJECTION = 81,
BADLY_STRUCTURED_APDU_REJECTION = 82,
INITIATOR_RELEASING_REJECTION = 83,
UNRECOGNIZED_LINKEDID_REJECTION = 84,
LINKED_RESPONSE_UNEXPECTED_REJECTION = 85,
UNEXPECTED_CHILD_OPERATION_REJECTION = 86,
MISTYPED_RESULT_REJECTION = 87,
UNRECOGNIZED_ERROR_REJECTION = 88,
UNEXPECTED_ERROR_REJECTION = 89,
MISTYPED_PARAMETER_REJECTION = 90,
NON_STANDARD = 100

SecurityError ::= ENUMERATED
    {unspecified (0),
     sequenceNumberViolated (1),
     timeStampViolated (2),
     pACViolated (3),
     sealViolated (4)}

See explanation below.

The CSTAUniversalFailure_t type is defined in TSAPI as type enum, while the ECMA-180 equivalent type UniversalFailure is defined as a CHOICE of several error class types including operationalErrors (defined as type Operations), stateErrors (defined as type StateIncompatibility), systemResourceErrors (defined as type SystemResourceAvailability), subscribedResourceAvailabilityErrors (defined as type SubscribedResourceAvailability), performanceErrors (defined as type PerformanceManagement), securityErrors (defined as type SecurityError), unspecifiedErrors (defined as type NULL), and nonStandardErrors (defined as type CSTAPrivateData). UniversalFailure is mapped in both the ECMA-180 to TSDI and TSDI to ECMA-180 direction. Most of the error types defined within the ECMA-180 error class types can be mapped one-to-one with the values specified for CSTAUniversalFailure_t. The exceptions are nonStandardErrors, and the values above 70 of CSTAUniversalFailure_t. In the ECMA-180 to TSDI direction, if a UniversalFailure is encoded as nonStandardErrors, then the corresponding CSTAUniversalFailure_t member will be set to NON_STANDARD and the associated private data will be treated as described in section {4.4.8.}. A value of GENERIC_UNSPECIFIED is used in the case where a UniversalFailure type is omitted (that is, when it is OPTIONAL). In the TSDI to ECMA-180 direction, if a CSTARouteEndRequest_t contains a value in the range 70 through 90, inclusive, for errorValue, then the corresponding UniversalFailure type element is encoded as unspecifiedErrors. If a value of NON_STANDARD is specified, then the request must be accompanied by non standard errors private data (of type CSTAPrivateData) in the TSAPI private data block. Other private data (see CSTACommonArguments mapping) may also be concatenated with the non standard errors private data, but the non standard errors private data must appear first. See section {4.4.8.} for additional information.
typedef enum CSTAEventCause_t 
{ 
    EC_NONE = -1, 
    EC_ACTIVE_MONITOR = 1, 
    EC_ALTERNATE = 2, 
    EC_BUSY = 3, 
    EC_CALL_BACK = 4, 
    EC_CALL_CANCELED = 5, 
    EC_CALL_FORWARD_ALWAYS = 6, 
    EC_CALL_FORWARD_BUSY = 7, 
    EC_CALL_FORWARD_NO_ANSWER = 8, 
    EC_CALL_FORWARD = 9, 
    EC_CALL_NOT_ANSWERED = 10, 
    EC_CALL_PICKUP = 11, 
    EC_CAMP_ON = 12, 
    EC_DEST_NOT_OBTAINABLE = 13, 
    EC_DO_NOT_DISTURB = 14, 
    EC_INCOMPATIBLE_DESTINATION = 15, 
    EC_INVALID_ACCOUNT_CODE = 16, 
    EC_KEY_CONFERENCE = 17, 
    EC_LOCKOUT = 18, 
    EC_MAINTENANCE = 19, 
    EC_NETWORK_CONGESTION = 20, 
    EC_NETWORK_NOT_OBTAINABLE = 21, 
    EC_NEW_CALL = 22, 
    EC_NO_AVAILABLE_AGENTS = 23, 
    EC_OVERRIDE = 24, 
    EC_PARK = 25, 
    EC_OVERFLOW = 26, 
    EC_RECALL = 27, 
    EC_REDIRECTED = 28, 
    EC_REORDER_TONE = 29, 
    EC_RESOURCES_NOT_AVAILABLE = 30, 
    EC_SILENT_MONITOR = 31, 
    EC_TRANSFER = 32, 
    EC_TRUNKS_BUSY = 33, 
    EC_VOICE_UNIT_INITIATOR = 34 
} CSTAEventCause_t;

EventCause ::= 
    ENUMERATED 
    {activeMonitor (1), 
    alternate (2), 
    busy (3), 
    callBack (4), 
    callCanceled (5), 
    callForwardAlways (6), 
    callForwardBusy (7), 
    callForwardNoAnswer (8), 
    callForward (9), 
    callNotAnswered (10), 
    callPickup (11), 
    campOn (12), 
    destNotObtainable (13), 
    doNotDisturb (14), 
    incompatibleDestination (15), 
    invalidAccountCode (16), 
    keyConference (17), 
    lockout (18), 
    maintenance (19), 
    networkCongestion (20), 
    networkNotObtainable (21), 
    newCall (22), 
    noAvailableAgents (23), 
    override (24), 
    park (25), 
    overflow (26), 
    recall (27), 
    redirected (28), 
    reorderTone (29), 
    resourcesNotAvailable (30), 
    silentMonitor (31), 
    transfer (32), 
    trunksBusy (33), 
    voiceUnitInitiator (34) }

The CSTAEventCause_t type is defined in TSAPI as type enum, while the ECMA-180 equivalent type EventCause is defined as type ENUMERATED. The possible values of CSTAEventCause_t and EventCause are nearly identical (that is, the only exception is that CSTAEventCause_t includes an EC_NONE value and EventCause does not) and map in a straightforward manner as shown in the table above. The mapping only occurs in the ECMA-180 to TSDI direction. If an EventCause type element is omitted (that is, when it is OPTIONAL), the corresponding CSTAEventCause_t type member will be set to EC_NONE.
4.4.7. Security Service

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No equivalent TSAPI definition exists. CSTA security services as defined in ECMA-179 &amp; ECMA-180 are not supported in the TSDI interface.</td>
<td>Seal ::= SEQUENCE (algorithmIdentifier OCTET STRING, keyIdentifier OCTET STRING, seal OCTET STRING)</td>
<td>See explanation below.</td>
</tr>
<tr>
<td>CSTASecurityData ::= SEQUENCE (messageSequenceNumber INTEGER OPTIONAL, timeStamp UTCTime OPTIONAL, privilegeAttributeCertificate PrivilegeAttributeCertificate OPTIONAL, seal Seal OPTIONAL)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The ECMA-180 CSTASecurityData type is used to carry information related to the CSTA Security Service. There is no applicable counterpart definition in TSAPI. The Telephony Server provides other mechanism for security. The Cserver does not generate CSTASecurityData (that is, it is always OPTIONAL) in any requests or responses send to the switch/driver, and always ignores any CSTASecurityData that may be present in any requests or responses received from the switch/driver.

4.4.8. Common Extensions and Private Data

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
</table>

The ECMA-180 CSTACommonArguments type is used to define the common arguments to CSTA requests and is defined as a sequence containing security (defined as type CSTASecurityData) and privateData (defined as SEQUENCE OF CSTAPrivateData). There is no equivalent TSAPI defined type. CSTACommonArguments is encoded only when a client request contains private data. The security element is never encoded (that is, it is OPTIONAL). The mapping of CSTACommonArguments requires the client application to send or receive an ASN.1/BER encoded sequence of CSTAPrivateData within the TSAPI private data block. The encoding shall be the same as that of the privateData element of CSTACommonArguments, including the initial tag of [1] (that is, context-specific 1). If the client request contains private data, and the vendor string matches the identifier in the priv_data_vendor argument supplied by the driver at registration time, then the private data will be encoded within the ECMA-180 request. The Cserver checks that the private data encoding supplied by the application is valid before passing the request to the PBX Driver. An invalid encoding will fail the entire request. If the TSAPI request does not contain private data or the driver registered with a NULL value of priv_data_vendor, then CSTACommonArguments is not encoded (that is, it is OPTIONAL in all requests except Escape Service). In the ECMA-180 to TSDI direction, any private data present within CSTACommonArguments in a received APDU is copied to the TSAPI private data block in the same format as defined above, and the vendor string is set to the identifier in the priv_data_vendor value supplied at registration time. If this value was NULL, then the private data is discarded and is not received by the client application.

typedef struct PrivateData_t { char vendor[32]; unsigned short length; char data[1]; } PrivateData_t; CSTAPrivateData ::= [APPLICATION 29] IMPLICIT SEQUENCE (manufacturer OBJECT IDENTIFIER, data ANY DEFINED BY manufacturer)
The `CDIOctetStringMap_t` structure contains 5 variables which control the mapping for each identifier type. The effect of direction (that is, some of the variables are only used in one mapping) are described in the following table. Note that in all cases, at most 4 octets in the identifier can be controlled by the PBX driver via the `monitorCrossRefID_map` parameter to the `cdiDriverRegister()` function at driver registration time. This parameter is a pointer to a structure of type `CDIMapping_info_t`, which contains 3 instances of a structure of type `CDIOctetStringMap_t`. These three structure instances named `callID_map`, `monitorCrossRefID_map`, and `routingCrossRefID_map`, define the mappings to be applied for the `call` element of the `ConnectionID` types, `MonitorCrossRefID` types, and `RoutingCrossRefID` types, respectively. The `CDIOctetStringMap_t` structure contains 5 variables which control the mapping for each identifier type. The effect of each mapping variable and the direction of significance (that is, some of the variables are only used in one mapping direction) are described in the following table. Note that in all cases, at most 4 octets in the identifier `OCTET STRING` are significant (that is, mapped).

<table>
<thead>
<tr>
<th>TSAPI C-Language Definition</th>
<th>ECMA-180 Defined Type (ASN.1 definition)</th>
<th>Mapping (C Lang. Type &lt;-&gt; ECMA-180 Type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No equivalent TSAPI type exists.</td>
<td>PrivateEventInfo ::= NULL -- The actual encoding of the private event is added here, replacing NULL -- with another valid ASN.1 type</td>
<td>See explanation below.</td>
</tr>
</tbody>
</table>

The `PrivateEventInfo` type is defined in ECMA-180 as type `NULL`, however, as indicated by the accompanying comment, it is intended to be replaced by any valid ASN.1 type. Note that since `PrivateEventInfo` is defined as the `EVENTINFO` (see `EVENT` macro definition in `ECMA-180`) of the CSTA private event, and that any EVENT can also conditionally contain CSTA private data, an ECMA-180 private event can contain both `PrivateEventInfo` and `CSTAPrivateData`. Since the `TSAPI` private status event definition only allows for a single block of CSTA private data to be returned to the application, the `PrivateEventInfo` and `CSTAPrivateData`, if any, are concatenated and returned as a single block of `TSAPI` private data (that is, `PrivateData_t`) to the application. The `PrivateEventInfo` always appears first in the buffer, even if it is `NULL`. It is left up to the application to discern the two blocks of data within the private data buffer. `PrivateEventInfo` can be any valid BER encoded ASN.1 type (see [X.208] and [X.209]).

### 4.4.9. Octet String to Long Identifier Mapping

The `call` element of the `ConnectionID` type, the `MonitorCrossRefID` type, and the `RoutingCrossRefID` type are all defined as the ASN.1 `OCTET STRING` type in ECMA-180. The equivalent types are declared as type `long` (that is, 32-bit integer) in `TSAPI`. The mapping from the ECMA-180 representation to the `TSAPI` representation, and vice-versa, can be controlled by the PBX driver via the `mapping_descriptor` parameter to the `cdiDriverRegister()` function at driver registration time. This parameter is a pointer to a structure of type `CDIMapping_info_t`, which contains 3 instances of a structure of type `CDIOctetStringMap_t`. These three structure instances named `callID_map`, `monitorCrossRefID_map`, and `routingCrossRefID_map`, define the mappings to be applied for the `call` element of the `ConnectionID` types, `MonitorCrossRefID` types, and `RoutingCrossRefID` types, respectively. The `CDIOctetStringMap_t` structure contains 5 variables which control the mapping for each identifier type. The effect of each mapping variable and the direction of significance (that is, some of the variables are only used in one mapping direction) are described in the following table. Note that in all cases, at most 4 octets in the identifier `OCTET STRING` are significant (that is, mapped).
<table>
<thead>
<tr>
<th>Mapping Variable</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>significant_octets</td>
<td>OCTET STRING-&gt;long</td>
<td>This variable indicates which octets are significant if the octet string exceeds 4 in length. A value of CDI_FIRST_FOUR indicates that the first 4 octets are used in mapping the identifier to a long integer, and a value of CDI_LAST_FOUR indicates that the last 4 octets of the OCTET STRING type are used in mapping the identifier to a long integer.</td>
</tr>
<tr>
<td>byte_order</td>
<td>both directions</td>
<td>This variable indicates the byte ordering of the OCTET STRING representation of an identifier. A value of CDI_MSB_FIRST indicates the first octet is most significant (that is, resides in the most significant byte of the long integer), and a value of CDI_LSB_FIRST indicates the first octet is the least significant (that is, resides in the least significant byte of the long integer).</td>
</tr>
<tr>
<td>min_length</td>
<td>long-&gt;OCTET STRING</td>
<td>This variable indicates the minimum length octet string to encode for the identifier. The value of this field must be between 1 and 4 inclusive and must be less than or equal to the value of the max_length variable.</td>
</tr>
<tr>
<td>max_length</td>
<td>long-&gt;OCTET STRING</td>
<td>This variable indicates the maximum length octet string to encode for the identifier. The value of this field must be between 1 and 4 inclusive and must be greater than or equal to the value of the min_length variable.</td>
</tr>
<tr>
<td>strip_null</td>
<td>long-&gt;OCTET STRING</td>
<td>This variable is a boolean that indicates whether high-order NULL bytes within the long are stripped prior to encoding the identifier as an OCTET STRING type. The field should be set to TRUE if stripping is desired, FALSE if it is not. The OCTET STRING encoding will never be stripped to less than the min_length variable.</td>
</tr>
</tbody>
</table>

### 4.4.10. Octet String to Char String Mapping

A number of CSTA types are defined as type OCTET STRING in ECMA-180 and type char string (that is, array of char) in TSAPI. The TSAPI definition follows the C-Language convention of using a NULL to terminate (that is, delimit the end of) a char string. These types are generally mapped by copying the string byte for byte from one representation to the other. However, two cases require special handling: when the OCTET STRING value exceeds the maximum length string defined in TSAPI, and when NULL values are embedded inside the OCTET STRING. In the former case, the mapping function will silently truncate the OCTET STRING to fit the maximum length string defined in TSAPI for a given type. This should not be a problem for most switch/driver implementations, as the TSAPI definitions are more than adequate in length to accommodate the required information. In the latter case, any embedded NULL is replaced by the two-character sequence "\0" (that is, backslash and ASCII '0'). This mapping also applies in the reverse direction, such that the sequence "\0" is converted to a NULL octet. A backslash can be escaped by another backslash if needed (that is, "\\0" yields two octets, \ and \0). A backslash followed by any character other than another backslash or '0' is not treated specially. The need for NULL octets in CSTA types should be very infrequent.
4.5. ROSE Error Mappings

CCITT Recommendation X.219/X.229 define a set of errors which can be exchanged in ROSE APDUs for certain types of errors detected either at the ROSE-provider or ROSE-user (that is, application) level. These errors will be mapped into TSDI format error codes according to the table below. A request generated by the application that is rejected by the driver will receive a `CSTAUniversalFailureConfEvent` event with the `error` field set as indicated in the TSAPI error code column below.

<table>
<thead>
<tr>
<th>ROSE Error Class</th>
<th>ROSE Error</th>
<th>TSAPI Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeneralProblem</td>
<td>recognisedAPDU</td>
<td>UNRECOGNIZED_APDU_REJECTION</td>
</tr>
<tr>
<td></td>
<td>mistypedAPDU</td>
<td>MISTYPED_APDU_REJECTION</td>
</tr>
<tr>
<td></td>
<td>badlyStructuredAPDU</td>
<td>BADLY_STRUCTURED_APDU_REJECTION</td>
</tr>
<tr>
<td>InvokeProblem</td>
<td>duplicateInvocation</td>
<td>DUPLICATE_INVOCATION_REJECTION</td>
</tr>
<tr>
<td></td>
<td>recognisedOperation</td>
<td>UNRECOGNIZED_OPERATION_REJECTION</td>
</tr>
<tr>
<td></td>
<td>mistypedArgument</td>
<td>MISTYPED_ARGUMENT_REJECTION</td>
</tr>
<tr>
<td></td>
<td>resourceLimitation</td>
<td>RESOURCE_LIMITATION_REJECTION</td>
</tr>
<tr>
<td></td>
<td>initiatorReleasing</td>
<td>INITIATOR_RELEASING_REJECTION</td>
</tr>
<tr>
<td></td>
<td>recognisedLinkId</td>
<td>UNRECOGNIZED_LINKEDID_REJECTION</td>
</tr>
<tr>
<td></td>
<td>linkedResponseUnexpected</td>
<td>LINKED_RESPONSE_UNEXPECTED_REJECTION</td>
</tr>
<tr>
<td></td>
<td>unexpectedChildOperation</td>
<td>UNEXPECTED_CHILD_OPERATION_REJECTION</td>
</tr>
<tr>
<td>ReturnResultProblem</td>
<td>recognisedInvocation</td>
<td>(not applicable)</td>
</tr>
<tr>
<td></td>
<td>resultResponseUnexpected</td>
<td>(not applicable)</td>
</tr>
<tr>
<td></td>
<td>mistypedResult</td>
<td>MISTYPED_RESULT_REJECTION</td>
</tr>
<tr>
<td>ReturnErrorProblem</td>
<td>recognisedInvocation</td>
<td>(not applicable)</td>
</tr>
<tr>
<td></td>
<td>errorResponseUnexpected</td>
<td>(not applicable)</td>
</tr>
<tr>
<td></td>
<td>recognisedError</td>
<td>UNRECOGNIZED_ERROR_REJECTION</td>
</tr>
<tr>
<td></td>
<td>mistypedError</td>
<td>UNEXPECTED_ERROR_REJECTION</td>
</tr>
<tr>
<td></td>
<td>mistypedParameter</td>
<td>MISTYPED_PARAMETER_REJECTION</td>
</tr>
</tbody>
</table>

The interpretation of the table entries for the `ReturnResultProblem` and `ReturnErrorProblem` ROSE error classes should be as follows. These error classes apply to Cserver rejections of ROSE return result or return error APDUs received by the Cserver from the PBX driver. Under the conditions associated with the specific ROSE error, the Cserver will return the error to the driver (via the RO-REJECT-U service) and at the same time return the TSAPI error code indicated in the table to the requesting application. A table entry of "(not applicable)" means that no TSDI message is generated, but the reject APDU is still sent to the driver.

4.6. Error Log Interface

The Tserver exports a standard function call interface to the PBX Driver so that the PBX Driver can log errors to the Telephony Server log file. A PBX Driver that registers with the Cserver and uses the CSDI interface should still use the
TSDI interface for error logging (that is, \texttt{tdiLogError()}). It should use the same \texttt{driverID} parameter value to call \texttt{tdiLogError()} that was returned by the \texttt{cdiDriverRegister()} function. For details on the TSDI error logging interface, see [TSDI].

The CSTA server will itself generate error log entries for the following error cases. The severity level of the error is indicated in brackets.

1. ROSE Protocol Errors [ERROR]
2. Unrecognized Replies or Requests from the PBX Driver [ERROR]
3. Reply Rejection from the PBX Driver [WARNING]
4. Invalid Data Received from the PBX Driver [ERROR]
5. Prolonged CSDI or TSDI Queue Congestion [ERROR]
6. Inability to Send/Receive APDU across interface [FATAL]
7. Unrecognized Identifier (that is, InvokeID, MonitorCrossRefID, RoutingCrossRefID) Received from the PBX Driver [ERROR]
8. Insufficient Buffer Space (that is, to hold request/reply) [ERROR]

The following "normal" events will cause AUDIT_TRAIL type error log entries to be made by the Cserver:

1. Driver Register/Unregister
2. Change in System Status reported by the PBX Driver

5. Compiling and Linking a Driver

5.1. Compiling and Linking For NetWare

The Telephony Server was 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 CSDI have the same memory layout in the Cserver and the PBX Driver.

The following is a variable from the makefile used to build the CSTA.NLM which shows the compiler options used.

\[
p_{-}wcc386opt = /w4 /zp1 /3s /zl /od
\]

The CSDI library routines defined in section 7 (that is, \texttt{cdiDriverRegister}, \texttt{cdiAllocBuffer}, etc.) are export by the Cserver NLM using the NetWare \texttt{export} linker directive. A driver must use the NetWare \texttt{import} directive in the linking phase of building. When the driver NLM is loaded, these CSDI functions will by dynamically linked to the driver NLM.

5.2. Compiling and Linking For Windows NT

The Telephony Server and Cserver were compiled using the Visual C/C++ 32 bit compiler, version 2.2 with the following significant options:

\begin{itemize}
  \item Calling Convention: \texttt{__cdecl}
  \item Use Run-Time Library: Multithreaded
\end{itemize}
Struct Member Alignment: 8 bytes

This means all drivers must be compiled with eight byte ordering so that the C-structures passed across the CSDI have the same memory layout in the Cserver and the PBX Driver. The following is part of the makefile used to build the Cserver which shows the compiler options used.

CPP_PROJ=nologo /MT /W3 /GX /YX /O2

The CSDI library routines defined in section 7 (that is, cdiDriverRegister, cdiAllocBuffer, etc.) are exported by the Cserver using the __declspec(dllexport) linker directive. A driver must use the __declspec(dllimport) directive in the linking phase of building. When the driver DLL is loaded, these CSDI functions will be dynamically linked to the driver DLL. A macro, CDILIBAPI, is used in the cdi.h header file that will take care of the importing as long as the cdi.h header file is included.

6. CSDI Coding Examples

6.1. Registering the Driver with the CSTA Server

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

6.1.1. NetWare Coding Example

/* main: */

main()
{
  int rc;
  CDIMappingInfo_t mappingInfo;
  CDIDriverCaps_t driverCaps;

  /*
   * Register an Unload cleanup function
   */
  signal(myAtUnload, SIGTERM);

  /*
   * Setup mapping info.
   */
```c
mappingInfo.callID_map.significant_octets = CDI_FIRST_FOUR;
mappingInfo.callID_map.byte_order = CDI_LSB_FIRST;
mappingInfo.callID_map.min_length = 2;
mappingInfo.callID_map.max_length = 4;
mappingInfo.callID_map.strip_null = TRUE;

mappingInfo.monitorCrossRefID_map.significant_octets = CDI_LAST_FOUR;
mappingInfo.monitorCrossRefID_map.byte_order = CDI_LSB_FIRST;
mappingInfo.monitorCrossRefID_map.min_length = 4;
mappingInfo.monitorCrossRefID_map.max_length = 4;
mappingInfo.monitorCrossRefID_map.strip_null = FALSE;

mappingInfo.routingCrossRefID_map.significant_octets = CDI_LAST_FOUR;
mappingInfo.routingCrossRefID_map.byte_order = CDI_LSB_FIRST;
mappingInfo.routingCrossRefID_map.min_length = 2;
mappingInfo.routingCrossRefID_map.max_length = 2;
mappingInfo.routingCrossRefID_map.strip_null = FALSE;

/*
 * Setup driver capabilities struct for driver register.
 */

driverCaps.alternateCall = 1; /* supported */
driverCaps.answerCall = 1; /* supported */
driverCaps.callCompletion = 0; /* not supported */
driverCaps.clearCall = 1; /* supported */
driverCaps.clearConnection = 1; /* supported */
driverCaps.conferenceCall = 1; /* supported */
driverCaps.consultationCall = 0; /* not supported */
driverCaps.divertCall deflect = 1; /* supported */
driverCaps.divertCall.pickup = 1; /* supported */
driverCaps.divertCall.group = 0; /* not supported */
driverCaps.holdCall = 1; /* supported */
driverCaps.makeCall = 1; /* supported */
```

```c
: /* note: setting of remaining fields not shown */

: *

/*
 * Register the driver for CSTA Services
 */

CstaCDIHandle = cdiDriverRegister( CDI_CSDI_VERSION_1,
CDI_PROTOCOL_VERSION_1,
"vendor_name",
"driver_name",
"service_name",
0,
CDI_CSTA_SECURITY,
NULL,
&mappingInfo,
```
"priv_data_vendor#1",
&driverCaps);

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

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

    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 CSDI, cleans up resources       */
/*                                                            */
/*****************************************************************
myAtUnload()
{
    int rc;

    /*
    * Unregister the CSTA Driver
    */

    if ( (CstaCDIHandle >= 0) &&
        ((rc = cdiDriverUnregister(CstaCDIHandle)) != CDI_SUCCESS))
    {
        tdiLogError(DRIVER_NAME, FATAL, ERR_NO, 0,
            "cdiDriverUnregister failed : %d",rc);
    }
    */
* Clean up any other remaining resources
 */
}
6.1.2. Windows NT Coding Example

******************************************************************************/
/* DllMain: */
/* This routine is called by NT when the dll is loaded or unloaded. */
/* */
/******************************************************************************/
__declspec(dllexport) BOOL WINAPI DllMain(HINSTANCE hinstDll,
    DWORD fdwReason,
    LPVOID lpvReserved)
{
    switch(fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        // Do minimal processing here
        // Threads spun off from here will
        // NOT execute until this function
        // call completes.
        break;

    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
        break;
    case DLL_PROCESS_DETACH:
        // The library is being freed. Any
        // cleanup should have been done by
        // the tdiStopDriver() function.
        default:
            break;
    }

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

tdiLogError(DRIVER_NM, TRACE, ERR_NO, 0,
            "tdiStartDriver has been called. \"\n            About to startup the <DRIVER_NAME>\"));
/*
* driverInitialize() does not return until the
* driver is completely up and running or if an
* error has occurred.
*/

completionCode = driverInitialize();

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

int driverInitialize()
{

    int rc;
    CDIMappingInfo_t mappingInfo;
    CDIDriverCaps_t driverCaps;
    DWORD dwThreadID;
    HANDLE completionCode;

    /*
    * Setup mapping info.
    */
    mappingInfo.callID_map.significant_octets = CDI_FIRST FOUR;
    mappingInfo.callID_map.byte_order = CDI_LSB FIRST;
    mappingInfo.callID_map.min_length = 2;
    mappingInfo.callID_map.max_length = 4;
    mappingInfo.callID_map.strip_null = TRUE;
    mappingInfo.monitorCrossRefID_map.significant_octets =
                CDI LAST FOUR;
    mappingInfo.monitorCrossRefID_map.byte_order = CDI_LSB FIRST;
    mappingInfo.monitorCrossRefID_map.min_length = 4;
    mappingInfo.monitorCrossRefID_map.max_length = 4;
    mappingInfo.monitorCrossRefID_map.strip_null = FALSE;
    mappingInfo.routingCrossRefID_map.significant_octets =
                CDI LAST FOUR;
    mappingInfo.routingCrossRefID_map.byte_order = CDI_LSB FIRST;
    mappingInfo.routingCrossRefID_map.min_length = 2;
    mappingInfo.routingCrossRefID_map.max_length = 2;
    mappingInfo.routingCrossRefID_map.strip_null = FALSE;

    /*
    * Setup driver capabilities struct for driver register.
    */
    driverCaps.alternateCall = 1; /* supported */
    driverCaps.answerCall = 1; /* supported */
    driverCaps.callCompletion = 0; /* not supported */
    driverCaps.clearCall = 1; /* supported */
driverCaps.clearConnection = 1; /* supported */
driverCaps.conferenceCall = 1; /* supported */
driverCaps.consulationCall = 0; /* not supported */
driverCaps.divertCall.deflect = 1; /* supported */
driverCaps.divertCall.pickup = 1; /* supported */
driverCaps.divertCall.group = 0; /* not supported */
driverCaps.holdCall = 1; /* supported */
driverCaps.makeCall = 1; /* supported */

: /* note: setting of remaining fields not shown */
:

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

CstaCDIHandle = cdiDriverRegister( CDI_CSDI_VERSION_1,
CDI_PROTOCOL_VERSION_1,
"vendor_name",
"driver_name",
"service_name",
0,
CDI_CSTA_SECURITY,
NULL,
&mappingInfo,
"priv_data_vendor#1",
&driverCaps);

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

:
:
:

/**
 * Begin Thread that will send the sanity message
 * to the Cserver every minute.
 */

completionCode = (HANDLE)_beginthreadex(NULL, // security
0, // stack size
sanity, // thread function
(LPVOID) 0, // argument for new thread
0, // creation flags
&dwThreadId); // address of returned thread ID

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

ThreadCount++; 

CloseHandle(completionCode); : ; : return(TRUE); 
} 

/******************************************************************************/ 
/* tdiStopDriver: */ 
/* */ 
/* This routine is called by the Tserver before it frees the */ 
/* driver library. This is the unload function of the Driver */ 
/* and should not return until the Driver is completely */ 
/* unloaded. If it encounters an error that prohibits it */ 
/* from unloading properly it should return an error (FALSE) */ 
/* from this function. */ 
/* */ 
/******************************************************************************/ 
__declspec(dllexport) BOOL tdiStopDriver() 
{ 
    int completionCode; 
    
    tdiLogError(DRIVER_NM, TRACE, ERR_NO, 0, 
    "tdiStopDriver has been called. \\
    About to unload the <DRIVER_NAME>"); 
    
    completionCode = driverUnload(); 
    
    if(completionCode == TRUE) 
    { 
        /* The driver has completely unloaded. It */ 
        /* can no longer handle client requests. */ 
        return(TRUE); 
    } 
    else 
    { 
        /* The driver has encountered an error and cannot */ 
        /* unload successfully. The driver library will */ 
        /* not be freed. */ 
        return(FALSE); 
    } 
} 

76   Issue 1.7   January 1997
/***/ ***/ driverUnload: Unregisters with the CSDI, cleans up resources */ ***/ int driverUnload() { ***/ int rc; ***/ 

/* */ * Tell Sanity thread we’re unloading */

NotUnloading = FALSE; ReleaseSemaphore(SanitySemaphore, 1, NULL); 

/* */ * Unregister the CSTA Driver */

if ((rc = cdiDriverUnregister(CstaCDIHandle)) != CDI_SUCCESS) 

    tdiLogError(DRIVER_NAME, FATAL, ERR_NO, 0, 
    "cdiDriverUnRegister failed : %d", rc); 
    return(FALSE); 
}

/* */ * Clean up any other remaining resources and wait for any threads 
* to die here... (While (ThreadCount > 0)...) */

if (FailedToCleanupResources) 

    return(FALSE); 
}
else 

    return(TRUE); 
}

WINAPI sanity(LPVOID lpThreadParameter)

while(NotUnloading) 

    cdiDriverSanity((CDIHandle_t *)lpThreadParameter); 
    TIMED_WAIT_FOR_OBJECT(SanitySemaphore, 60000);
6.2. Receiving a CSTA Request

This section contains a coding example for receiving a CSTA request (that is, ECMA-180) APDU from the Cserver. In this example, the PBX Driver decodes the received APDU using a commercially available ASN.1 compiler and encoder/decoder product. This step is not required if the switch can accept the APDU directly. Note also that it may be necessary to call the decoder a second time to decode the ECMA-180 APDU, which is actually "wrapped" by the ROSE APDU. The details of this second decoding are not shown in this example. This example applies to both NetWare and Windows NT.

```c
char  *pdu;  /* CSDI buffer pointer */
long  pdu_len; /* APDU length */
ROSEapdus *outbuf; /* decoded APDU pointer */
long  out_len;  /* decoded length */
char  errmsg[ERR_MSG_LENGTH];  /* error message buffer */
extern void *roapdus;  /* decoder control table */

/*
 * Call receive routine to get APDU.
 */
if ((rc = cdiReceiveFromCSTA(CstaCDIHandle, &pdu)) != CDI_SUCCESS) {
    tdiLogError(DRIVER_NAME, FATAL, ERR_NO, 0,
                 "cdiReceiveFromCSTA failed : %d", rc);
    return;
}

/*
 * Call the ASN.1 decoder to decode the APDU.
 */
rc = decode(NULL, &pdu_num, &pdu, &pdu_len, &outbuf, &out_len,
            roapdus, BUFFER_PROVIDED, errmsg);

/*
 * Process the ROSE/ECMA-180 APDU here (details not shown).
 */
:
:
:

/*
 * Free the CSDI buffer.
 */
if ((rc = cdiFreeBuffer(CstaCDIHandle, pdu)) != CDI_SUCCESS) {
    tdiLogError(DRIVER_NAME, ERROR, ERR_NO, 0,
                 "cdiFreeBuffer failed : %d", rc);
```
6.3. Sending a CSTA Response

This section contains a coding example for sending a CSTA response (that is, ECMA-180) APDU to the Cserver. In this example, the PBX Driver encodes the APDU using a commercially available ASN.1 compiler and encoder/decoder product. This step is not required if the switch can generate the APDU directly. Note that in order to simplify this example, the APDU buffer is allocated using the maximum buffer size allowed (that is, CDI_MAX_BUFFER_SIZE). In an actual implementation, the PBX Driver should allocate the actual size buffer needed.

This example applies to both NetWare and Windows NT.

```c
char *pdu;               /* CSDI buffer pointer */
long pdu_len;            /* APDU length */
ROSEapdus * inbuf;      /* unencoded APDU pointer */
char errmsg[ERR_MSG_LENGTH]; /* error message buffer */
CDIBuf_flag_t buf_flag;  /* buffer alloc flags */
extern void *ropadus;    /* encoder control table */

/*
 * Get a CSDI buffer.
 */
pdu_len = CDI_MAX_BUFFER_SIZE;
rc = cdiAllocBuffer(CstaCDIHandle, &pdu, pdu_len, &buf_flag);
if (rc != CDI_SUCCESS)
{
    tdiLogError(DRIVER_NAME, ERROR, ERR_NO, 0,
                "cdiFreeBuffer failed : %d", rc);
    return;
}

/*
 * Examine buf_flag here and invoke flow control if necessary.
 */
if (buf_flag & CDI_EXCEED_HIWATER_MARK)
{
    /* invoke flow control to switch */
    :
    :
}

/*
 * Call the ASN.1 encoder to encode the APDU.
 */
rc = encode(NULL, ROSEapdus_PDU, inbuf, &pdu, &pdu_len,
            ropadus, BUFFER_PROVIDED, errmsg);

/*
 * Send the APDU to the CSTA Server.
 */
rc = cdiSendToCSTA(CstaCDIHandle, pdu, CDI_NORMAL_MESSAGE)
if (rc != CDI_SUCCESS)
{
    tdiLogError(DRIVER_NAME, ERROR, ERR_NO, 0,
```
"cdiSendToCSTA failed : %d", rc);
    return;
}
7. CSTA Services Driver Interface Manual Pages

The following manual pages describe the function call interface between the PBX Driver and the CSTA Server.

7.1. cdiDriverRegister ( )

This function allows a driver (PBX or other) to register itself with the Cserver. It registers its name with the Cserver, specifies some tags that will be maintained by the Cserver for maintenance queries, requests parameters that specify the memory allocation limits that will be imposed on the Cserver and PBX Drivers for this interface, and passes information needed by the Cserver to perform certain mappings of data contained in service requests/responses and to respond to clients about which CSTA services the driver supports.

Syntax

```c
#include <cdi.h>

CDIHandle_t cdiDriverRegister ( 
    CDIVersion_t csdi_version, /* INPUT */
    CDIProtocol_t protocol_version, /* INPUT */
    const char *vendor_name, /* INPUT */
    const char *driver_name, /* INPUT */
    const char *service_name, /* INPUT */
    CDISecurity_t driver_security, /* INPUT */
    const CDIBuf_info_t *buffer_descriptor, /* INPUT */
    const CDIMapping_info_t *mapping_descriptor, /* INPUT */
    const char *priv_data_vendor, /* INPUT */
    const CDIDriverCaps_t *driver_caps); /* INPUT */
```

Parameters

`csdi_version`

This parameter is set to the version of the CSDI with which the driver will function. This parameter is mandatory. The registration will fail if this parameter is set to an invalid version. Currently only one version of the CSDI exists. This field should be set to **CDI_CSDI_VERSION_1**.

`protocol_version`

This parameter is set to the version of the CSTA protocol with which the driver will function. This parameter is mandatory. The registration will fail if this parameter is set to an invalid version. Currently only one version of the CSTA protocol (that is, June 1992) is supported. This field should be set to **CDI_PROTOCOL_VERSION_1**.

`vendor_name`

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

`driver_name`
This is the NULL terminated ASCII string that the driver will provide to the Cserver to identify the driver for 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 CDI_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_name**

This is the NULL terminated ASCII string that the driver will provide to the Cserver 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 CDI_MAX_SERVICE_NAME. The service_name must be unique for every registration done by a driver (that is, for every unique driver_name all service_name must be unique.

**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:

- **CDI_CSTA_SECURITY** System 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.

- **CDI_LOGIN_SECURITY** System 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.

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

**buffer_descriptor**

This is a pointer to a buffer descriptor structure containing information about the memory that can be allocated for this CSTA Services Driver Interface. A NULL pointer will use the default values listed below for each element of the buffer descriptor. The values passed here are also used to specify the memory allocation limits for the TSDI interface between the Cserver and Tserver. This parameter only applies for the first registration by a PBX driver on any given Tserver. The supplied values are saved in the Tserver's database during this first registration. On all subsequent registrations, the database values will override those specified in the `cdiDriverRegister()` call. The memory parameters can be changed via the TSA application while the PBX driver is registered.

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 */  
    unsigned long lowater_mark; /* Low water mark for buffer 
    * allocation on this interface */  
  ) CDIBuf_info_t;

mapping_descriptor

This is a pointer to a mapping info structure containing the information about how to map call, monitor cross 
reference, and routing cross reference identifiers between the types defined by ECMA-180 and their equivalent 
types in TSAPI. This parameter is mandatory and cannot be NULL.

typedef struct 
{  
    CDIOctetStringMap_t callID_map; /* Specifies mapping 
    * parameters for call identifiers */  
    CDIOctetStringMap_t monitorCrossRefID_map; /* Specifies 
    * mapping parameters for 
    * monitor cross reference 
    * identifiers */  
    CDIOctetStringMap_t routingCrossRefID_map; /* Specifies 
    * mapping parameters for 
    * routing cross reference 
    * identifiers */  
  ) CDIMapping_info_t;

The CDIOctetStringMap_t structure is defined as follows:

typedef struct 
{  
    unsigned char significant_octets; /* Indicates which octets 
    * (CDI_FIRST_FOUR or CDI_LAST_FOUR) 
    * are significant. */  
    unsigned char byte_order; /* Indicates whether octets are 
    * encoded MSB first or LSB first */  
    unsigned char min_length; /* Indicates minimum length octet 
    * string that should be encoded */  
    unsigned char max_length; /* Indicates maximum length octet 
    * string that should be encoded */  
    Boolean_t strip_null; /* Indicates whether NULL bytes in 
    * long int should be stripped */  
  ) CDIOctetStringMap_t;

priv_data_vendor
This is the NULL terminated ASCII string that specifies the identifier to be associated with the vendor’s private data (that is, the string used to compare against or set the vendor field of the PrivateData_t type). This parameter may be set to NULL if private data is not supported by the switch/driver. The identifier can be optionally followed by a '#' character and a list of private data version numbers to be used by the Cserver to implement private data version negotiation with TSAPI clients opening streams to the registering PBX driver. The version list can be specified as a list of ':' (that is, colon) separated numeric ASCII version numbers or version ranges indicated by the '-' (that is, hyphen) character. For example, private data versions 1 and 3 through 5 could be specified as "1:3-5". The private data identifier must not exceed the maximum length of CDI_MAX_PRIV_DATA_VENDOR. Also, because the '#' character is used as a delimiter, it cannot be included in a private data identifier string. If the version list is omitted, a default list of "1" (that is, version 1 only) is assumed.

**driver_caps**

This is a pointer to a driver capabilities structure containing information about which CSTA services the driver supports. It is used by the Cserver to reply to client application cstaGetAPICaps() requests. This parameter is mandatory and cannot be NULL.

```c
typedef struct {
    short alternateCall;
    short answerCall;
    short callCompletion;
    short clearCall;
    short clearConnection;
    short conferenceCall;
    short consultationCall;
    struct {
        short deflect;
        short pickup;
        short group;
    } divertCall;
    short holdCall;
    short makeCall;
    short makePredictiveCall;
    struct {
        short msgWaiting;
        short doNotDisturb;
        short forward;
        short lastDialedNumber;
        short deviceInfo;
        short agentState;
    } queryDevice;
    short reconnectCall;
    short retrieveCall;
    struct {
        short msgWaiting;
        short doNotDisturb;
        short forward;
        short agentState;
    } setFeature;
    short transferCall;
    short eventReport;
    short callClearedEvent;
    short conferencedEvent;
    short connectionClearedEvent;
} driver_caps;
```
short deliveredEvent;
short divertedEvent;
short establishedEvent;
short failedEvent;
short heldEvent;
short networkReachedEvent;
short originatedEvent;
short queuedEvent;
short retrievedEvent;
short serviceInitiatedEvent;
short transferredEvent;
short forwardingEvent;
short messageWaitingEvent;
short notReadyEvent;
short readyEvent;
short workNotReadyEvent;
short workReadyEvent;
short backInServiceEvent;
short outOfServiceEvent;
struct {
    short driverInvoked;
    short applInvoked;
} privateEvent;
short routeRequestEvent;
short reRouteRequest;
short routeSelectRequest;
short routeUsedRequest;
struct {
    short driverInvoked;
    short applInvoked;
} routeEndRequest;
struct {
    short driverInvoked;
    short applInvoked;
} escapeService;
struct {
    short driverInvoked;
    short applInvoked;
} systemStatus;
struct {
    short device;
    short call;
    short callsViaDevice;
} monitorStart;
short changeMonitorFilter;
struct {
    short driverInvoked;
    short applInvoked;
} monitorStop;
short snapshotDevice;
short snapshotCall;
} CDIDriverCaps_t;

Return Values

This function returns a driverID on success that must be used in all subsequent function calls by the Cserver and the driver to identify this specific Cserver-Driver interface. The driverID is guaranteed to be a positive integer.
and has the same value as that returned by the Tserver when the Cserver registers with the Tserver on behalf of
the driver using the `tdiDriverRegister()` function. On failure this function returns one of the following
(negative) values:

**CDI_ERR_DUP_DRVR**
This error indicates that the combination of `vendor_name` and `service_name` provided has already been registered with the Tserver.

**CDI_ERR_NO_MEM**
This error indicates the CSDI was unable to allocate memory from the Operating System.

**CDI_ERR_MAX_DRVR**
This error indicates that the maximum number of registered drivers, `CDI_MAX_REGISTRATIONS`, has been reached, or that the maximum number of Tserver registered drivers, `TDI_MAX_REGISTRATIONS`, has been reached.

**CDI_ERR_EINVAL**
This error indicates that an invalid parameter was specified for the `cdiDriverRegister()` function call.

**CDI_ERR_BAD_VERSION**
This error indicates that an invalid version number was supplied in the `csdi_version` parameter.

**CDI_ERR_BAD_SECURITY**
This error indicates that an invalid value was supplied in the `driver_security` parameter.

**CDI_ERR_BAD_SRVC_NAME**
This error indicates that an invalid value was supplied in the `service_name` parameter.

**CDI_ERR_BAD_VENDOR_NAME**
This error indicates that an invalid value was supplied in the `vendor_name` parameter.

**CDI_ERR_BAD_DRVR_NAME**
This error indicates that an invalid value was supplied in the `driver_name` parameter.

**CDI_ERR_BAD_PROTOCOL**
This error indicates that an invalid value was supplied in the `protocol_version` parameter.
CDI_ERR_BAD_PRIVDATA_VNDR

This error indicates that an invalid value was supplied in the
priv_data_vendor parameter.

CDI_ERR_BAD_MAPPING_DESC

This error indicates that an invalid value was supplied in the
mapping_descriptor parameter.

CDI_ERR_BAD_DRVRCAPS

This error indicates that an invalid value was supplied in the
driver_caps parameter.

CDI_ERR_ESYS

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

Comments

This function is issued by the driver to set up a communication path with the Cserver, identify the name that will
be advertised by the Tserver (the name is generated from the vendor_name and service_name parameters),
specify the maximum amount of memory that may be used for message buffers used to exchange messages
between the Cserver and the drivers for this communication path, and to pass information needed by the Cserver
to perform certain mappings of data contained in service requests/responses and to respond to clients about
which CSTA services the driver supports. Both the Cserver and the driver must allocate message buffers from
the CSDI routines to send a message across this CSTA Services Driver Interface.

When a PBX Driver registers with the CSDI for the first time, it must specify the maximum amount of memory
that can be allocated for message buffers by the driver and the Cserver for this communication path. Each
message buffer allocated by the driver or the Cserver from the CSDI will include a (12 byte) header that will be
used to implement the monitoring and queuing of messages. (The message buffer header should not be
accessed by the driver or the Cserver; it is used by the CSDI routines.) This header is not charged to the space
allocated in the CSDI via the max_bytes field defined below. Also note that, as indicated previously, database
stored memory allocation parameters for the associated driver, if any, will override those passed in the
cdiDriverRegister() function call.
The structure of type CDIBuf_info_t is defined as follows:

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

Field definitions:

**max_bytes**

A non-negative integer indicating the maximum amount of memory that can be allocated by the Cserver and the drivers for message buffers used on this communication path between the driver and Cserver. The `cdiAllocBuffer()` 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 `CDI_MAX_BYTES_ALLOCATED`. The minimum value for `max_bytes` is `CDI_MIN_BYTES_ALLOCATED`.

**hiwater_mark**

A non-negative integer indicating a high water mark for the memory allocated for this CSTA Services Driver Interface. When the amount of memory allocated for this interface exceeds the high water mark, the `cdiAllocBuffer()` routine will return the buffer to the "caller" (if a memory block can be allocated from the Operating System), 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 `CDI_BUFFER_HI_WATER_MARK`.

**lowater_mark**

A non-negative integer indicating a low water mark for the memory allocated for this CSTA Services Driver Interface. When the amount of memory allocated for this interface exceeds the low water mark, the `cdiAllocBuffer()` routine will return the buffer to the "caller" (if a memory block can be allocated from the Operating System), and include an indication that the low water mark has been exceeded. If the `buffer_descriptor` parameter is a NULL pointer, this value will default to `CDI_BUFFER_LO_WATER_MARK`.

When a PBX Driver registers with the CSDL, it must specify how call identifiers, monitor cross reference identifiers, and routing cross reference identifiers (defined in CSTA) are translated between the ECMA-180 defined types and the TSAPI defined types for these identifiers. ECMA-180 defines all three identifiers as ASN.1 OCTET STRING types. TSAPI defines all three identifiers as long (32-bit) integers. For each identifier, a set of translation parameters are supplied in the CDIMapping_info_t structure.
The structure of type `CDIMapping_info_t` is defined as follows:

```c
typedef struct
{
    CDIOctetStringMap_t callID_map;
    CDIOctetStringMap_t monitorCrossRefID_map;
    CDIOctetStringMap_t routingCrossRefID_map;
} CDIMapping_info_t;
```

Field definitions:

`callID_map`
A structure of type `CDIOctetStringMap_t` that specifies the translation or mapping parameters to be used for mapping between the ASN.1 OCTET STRING representation of call identifier in **ECMA-180** and the long integer representation of call identifier in **TSAPI**.

`monitorCrossRefID_map`
A structure of type `CDIOctetStringMap_t` that specifies the translation or mapping parameters to be used for mapping between the ASN.1 OCTET STRING representation of monitor cross reference identifier in **ECMA-180** and the long integer representation of monitor cross reference identifier in **TSAPI**.

`routingCrossRefID_map`
A structure of type `CDIOctetStringMap_t` that specifies the translation or mapping parameters to be used for mapping between the ASN.1 OCTET STRING representation of routing cross reference identifier in **ECMA-180** and the long integer representation of routing cross reference identifier in **TSAPI**.

The structure of type `CDIOctetStringMap_t` is defined as follows:

```c
typedef struct
{
    unsigned char significant_octets;
    unsigned char byte_order;
    unsigned char min_length;
    unsigned char max_length;
    Boolean_t strip_null;
} CDIOctetStringMap_t;
```
Field definitions:

**significant_octets**
This field is used to indicate which octets are significant if the octet string exceeds 4 in length. It is only used when translating from the OCTET STRING representation of an identifier to the long integer representation. This field should be set to CDI_FIRST_FOUR to indicate that the first 4 octets of the OCTET STRING type should be used in mapping the identifier to a long integer, and CDI_LAST_FOUR to indicate that the last 4 octets of the OCTET STRING type should be used in mapping the identifier to a long integer.

**byte_order**
This field is used to indicate the byte ordering of the OCTET STRING representation of an identifier. The field should be set to CDI_MSB_FIRST if the first octet is most significant (that is, is placed in the most significant byte of the long integer), and CDI_LSB_FIRST if the first octet is the least significant (that is, is placed in the least significant byte of the long integer). This field is used for translating identifiers in both directions.

**min_length**
This field is used to indicate the minimum length octet string to encode for the identifier. It is only used when mapping a long integer representation of the identifier to the octet string representation. The value of this field must be between 1 and 4 inclusive and must be less than or equal to the value supplied in the **max_length** field.

**max_length**
This field is used to indicate the maximum length octet string to encode for the identifier. It is only used when mapping a long integer representation of the identifier to the octet string representation. The value of this field must be between 1 and 4 inclusive and must be greater than or equal to the value supplied in the **min_length** field.

**strip_null**
This field is a Boolean type used to indicate whether high-order (that is, most significant) NULL bytes within the long integer representation of the identifier are stripped prior to encoding the identifier as an OCTET STRING type. The field should be set to TRUE if stripping is desired, FALSE if it is not. The OCTET STRING encoding will never be stripped to less than the value indicated by **min_length**.

When a PBX Driver registers with the CSDL, it may specify a **priv_data_vendor** identifier value for the purpose of passing CSTA private data within CSTA requests, responses, and events. This identifier is a string that is compared with the **vendor** field of the PrivateData_t structure in a request received from a client application. If the two strings match, then the private data is encoded within the request sent to the driver, or else it is discarded (that is, its format is unknown and hence cannot be encoded in a ECMA-180 APDU). Private data encoded in requests, responses, and events received from the driver are passed to the client application in the TSAPI private data block and the **vendor** field of the PrivateData_t structure is set to the supplied **priv_data_vendor** identifier value. If a value of NULL (or NULL string) is supplied for
priv_data_vendor, then no private data is passed between the driver and client application in either direction. The priv_data_vendor parameter may also include a list of private data version numbers to be used to negotiate private data versions with the client application on behalf of the PBX driver. This TSAPI-specified version negotiation takes place during the processing of an open stream request from the client to the Cserver.

Finally, when a PBX Driver registers with the Cserver it must supply information about which CSTA services it supports. This information, passed in the driver_caps parameter, is used to reply to client application cstaGetAPICaps() requests. The information is not used by the Cserver to block requests not supported by the driver from reaching the driver. The driver must be prepared to reject services it does not support regardless of the values supplied in the driver_caps parameter during driver registration.

The structure of type CDIDriverCaps_t is defined as follows:

```c
typedef struct CDIDriverCaps_t {
    short alternateCall;
    short answerCall;
    short callCompletion;
    short clearCall;
    short clearConnection;
    short conferenceCall;
    short consultationCall;
    struct {
        short deflect;
        short pickup;
        short group;
    } divertCall;
    short holdCall;
    short makeCall;
    short makePredictiveCall;
    struct {
        short msgWaiting;
        short doNotDisturb;
        short forward;
        short lastDialedNumber;
        short deviceInfo;
        short agentState;
    } queryDevice;
    short reconnectCall;
    short retrieveCall;
    struct {
        short msgWaiting;
        short doNotDisturb;
        short forward;
        short agentState;
    } setFeature;
    short transferCall;
    short eventReport;
    short callClearedEvent;
    short conferencedEvent;
    short connectionClearedEvent;
    short deliveredEvent;
    short divertedEvent;
    short establishedEvent;
} CDIDriverCaps_t;
```
short failedEvent;
short heldEvent;
short networkReachedEvent;
short originatedEvent;
short queuedEvent;
short retrievedEvent;
short serviceInitiatedEvent;
short transferredEvent;
short callInformationEvent;
short doNotDisturbEvent;
short forwardingEvent;
short messageWaitingEvent;
short loggedOnEvent;
short loggedOffEvent;
short notReadyEvent;
short readyEvent;
short workNotReadyEvent;
short workReadyEvent;
short backInServiceEvent;
short outOfServiceEvent;
struct {
    short driverInvoked;
    short applInvoked;
} privateEvent;
short routeRequest;
short reRouteRequest;
short routeSelectRequest;
short routeUsedRequest;
struct {
    short driverInvoked;
    short applInvoked;
} routeEndRequest;
struct {
    short driverInvoked;
    short applInvoked;
} escapeService;
struct {
    short driverInvoked;
    short applInvoked;
} systemStatus;
struct {
    short device;
    short call;
    short callsViaDevice;
} monitorStart;
short changeMonitorFilter;
struct {
    short driverInvoked;
    short applInvoked;
} monitorStop;
short snapshotDevice;
short snapshotCall;
} CDIDriverCaps_t;
Each field of CDIDriverCaps_t is a Boolean value indicating whether the associated service is supported or not. A value of zero indicates the service is not supported, while a non-zero value indicates the service is supported and the value indicates the CSTA version that is supported for that service. Currently, the version number should be set to 1 for all supported services. Some services are broken down by applicable service options to accommodate the fact that TSAPI defines them as separate services. If a service is defined to be bidirectional by ECMA-179, then the structure contains a field for each direction (that is, driver invoked or application invoked).
7.2. cdiDriverUnregister ( )

This function allows a driver (PBX or other) to unregister itself with the Cserver. It must use the driver_id that was returned by the cdiDriverRegister() routine.

Syntax

```c
#include <cdi.h>
CDIReturn_t cdiDriverUnregister (  
       CDIHandle_t driverID); /*INPUT*/
```

Parameters

`driverID`

This is the unique identification number given to the driver when it registered with the Cserver.

Return Values

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

- **CDI_ERR_BAD_DRVRID** This error indicates that the `driverID` specified in the cdiDriverUnregister() function is not valid.
- **CDI_ERR_ESYS** This error indicates that some form of system error has occurred. When this occurs the CSDI will place an entry in the Error Log.

Comments

This routine is exported from the Cserver for use by Drivers that import the CSDI routines. It will cause the Cserver to delete the Driver List entry that was created when the driver originally registered with the Cserver via the cdiDriverRegister() routine. All memory that was allocated for this CSTA Services Driver Interface will be given back to the Operating System. All messages in the queues will be removed and the queues will be deleted. If the driver unloads before calling this routine, the Cserver will attempt to deallocate all of the resources associated with this CSTA Services Driver Interface. The Cserver will unregister with the Tserver on behalf of the driver, and all resources and messages associated with the Telephony Services Driver Interface will be deallocated by the Tserver. This routine will not block, but only signal the Cserver that the driver wishes to unregister. After the routine returns, the Cserver performs the actions specified above.
7.3. cdiAllocBuffer()

This function is issued by the Cserver or the driver to allocate a buffer for sending a message across the CSTA Services Driver Interface.

Syntax

```c
#include <cdi.h>

CDIReturn_t cdiAllocBuffer (
    CDIHandle_t driverID, /* INPUT */
    char **bufptr, /* OUTPUT */
    unsigned int length, /* INPUT */
    CDIBuf_flag_t *buf_flag); /* OUTPUT */
```

Parameters

**driverID**

This is the value of the handle returned by the `cdiDriverRegister()` function call. This handle uniquely identifies the interface between the Cserver and the driver.

**bufptr**

This parameter is set to point to the start of the buffer returned by the `cdiAllocBuffer()` function call. If the `cdiAllocBuffer()` 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. If the `cdiAllocBuffer()` routine is successful, this function will return a block of data that is at least `length` bytes long. *(The CSDI will allocate a block of memory that is larger than length. The first 12 bytes of this memory block will be used by the CSDI to implement the message queues, however, this 12 bytes should not be included in the length field. The CSDI 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 `cdiAllocBuffer()` routine to provide information on the amount of memory allocated by the Cserver or the PBX Driver for this interface. The `buf_flag` parameter will be set to indicate the following conditions when they occur:

**CDI_EXCEED_LOWATER_MARK**

The current amount of memory allocated for message buffers (by the Cserver and the PBX Driver) is greater than the low water mark (`lowater_mark`) specified in the `cdiDriverRegister()` function.
CDI_EXCEED_HIWATER_MARK  The current amount of memory allocated for message buffers (by the Cserver and the PBX Driver) is greater than the high water mark (hiwater_mark) specified in the cdiDriverRegister() function.

CDI_EXCEED_MAX_BYTES  The current amount of memory allocated for message buffers (by the Cserver and the PBX Driver) plus the requested buffer length is greater than the maximum number of bytes (max_bytes) specified in the cdiDriverRegister() function. This will only be returned on a failure.

Return Values

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

CDI_ERR_BAD_DRVRID  This error indicates that the driverID specified in the cdiAllocBuffer() function is not valid.

CDI_ERR_NO_MEM  This error indicates the CSDI was unable to allocate the requested memory from the Operating System.

CDI_ERR_BADLENGTH  This error indicates that the requested length is greater than CDI_MAX_BUFFER_SIZE (NetWare 3.x only).

CDI_ERR_NO_BUFFERS  This error indicates that the Cserver and the driver have (together) allocated more memory for message buffers than allowed for this CSTA 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 CSDI.

CDI_ERR_ESYS  This error indicates that some form of system error has occurred. When this occurs the CSDI will place an entry in the Error Log.
Comments

The `cdiAllocBuffer()` function provides a buffer to the Cserver or the PBX Driver that can be used to send a message across the CSTA Services Driver Interface. The buffers are allocated from the Operating System if the current amount of memory allocated for this interface is less than the maximum specified at driver registration time. The driver is responsible for setting the maximum bytes allowed for this interface during the `cdiDriverRegister()` routine. If the driver has allocated a message buffer via the `cdiAllocBuffer()` routine, the driver is responsible for dealing with the buffer by either sending the buffer back to the Cserver through the CSDI via the `cdiSendToCSTA()` function, in which case the Cserver is responsible for freeing the buffer, or freeing the buffer via the `cdiFreeBuffer()` function. If the Cserver has allocated a message buffer via the `cdiAllocBuffer()` routine, the Cserver is responsible for freeing the buffer back to the CSTA Services Driver Interface either by successfully sending the buffer to the driver via the `cdiSendToDriver()` function, or freeing the buffer via the `cdiFreeBuffer()` function.

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

Memory allocated for message buffers via the `cdiAllocBuffer()` routine should not be directly freed back to the Operating System by the Cserver or the drivers; the messages should be released back to the CSTA Services Driver Interface via the `cdiFreeBuffer()` routine.

Warning

The Cserver and drivers are not guaranteed to receive a message buffer via the `cdiAllocBuffer()` routine even though the current memory allocated is less than `max_bytes` specified by the driver in the `cdiDriverRegister()` routine. The Operating System may not have the resources at this time to fulfill the memory allocation request.

Driver Notes

The PBX Driver is responsible for issuing a `cdiDriverRegister()` function call to specify the maximum number of bytes that can be allocated for message buffers by the Cserver or the driver for this interface. The `driverID` returned by the `cdiDriverRegister()` routine must be used to allocate buffers that will be used across this interface. The driver should monitor the `buf_flag` parameter to determine if the memory allocation limit is sized appropriately and to determine when some form of flow control is required. The driver is not responsible for the memory resources allocated by `cdiAllocBuffer()` routine since they are allocated from the Operating System. The CSTA Services Driver Interface is responsible for freeing these memory resources. The driver is responsible for giving the memory resources back to the CSTA Services Driver Interface via the `cdiFreeBuffer()` routine, or sending the buffer to the Cserver via the `cdiSendToCSTA()` routine.
Cserver Notes

The \textit{driverID} must be used to allocate buffers that will be used across this CSTA Services Driver Interface. The Cserver is responsible for giving the memory resources back to the CSTA Services Driver Interface via the \texttt{cdiFreeBuffer()} routine, or sending the buffer to the driver via the \texttt{cdiSendToDriver()} routine. The CSTA Services Driver Interface is part of the Cserver, and the memory allocated for the CSTA Services Driver Interface must be freed before the Cserver can unload.
7.4. `cdiFreeBuffer()`

This function is issued by the Cserver or PBX Driver to free a buffer that was previously allocated to transmit a message across the CSTA Services Driver Interface.

Syntax

```c
#include <cdi.h>

CDIReturn_t cdiFreeBuffer(
    CDIHandle_t driverID, /* INPUT */
    char *bufptr); /* INPUT */
```

Parameters

- **driverID**
  
  This is the value of the handle returned by the `cdiDriverRegister()` function call. This handle uniquely identifies the CSTA Services Driver Interface.

- **bufptr**
  
  This parameter is a pointer to the start of the buffer returned by the `cdiAllocBuffer()` function call (for either the Cserver or the driver), a `cdiReceiveFromDriver()` function call (for the Cserver), or a `cdiReceiveFromCSTA()` function call (for the driver). After the `cdiFreeBuffer()` routine completes, the "caller" should no longer access the buffer.

Return Values

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

- **CDI_ERR_BAD_DRVRID**
  
  This error indicates that the `driverID` specified in the `cdiFreeBuffer()` function is not valid.

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

- **CDI_ERR_NOT_YOUR_BUFFER**
  
  This error indicates that the `driverID` specified did not match the `driverID` stored with this CSDI Buffer when the buffer was created via the `cdiAllocBuffer()` call. A driver is only allowed to free CSDI buffers that were originally allocated for this driver. Note: All CSDI buffers allocated by a Cserver to be sent across the CSDI to a driver are allocated with that Driver’s ID.

  When this occurs the CSDI will place an entry in the Error Log.
CDI_ERR_ESYS

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

Comments

The cdifreeBuffer() function returns a buffer to the Operating System that was previously allocated to send a message between the driver and the Cserver.

Warning

Memory allocated for message buffers via the cdiallocBuffer() routine should not be directly freed back to the Operating System by the Cserver or the drivers; the messages should be released back to the CSTA Services Driver Interface via the cdifreeBuffer() routine.

Memory allocated from the Operating System directly may not be released back to the OS by the cdifreeBuffer() routine.

Driver Notes

The PBX Driver is responsible for issuing a cdidriverRegister() function call to specify the maximum number of bytes that can be allocated for message buffers by the Cserver or the driver for this interface. The driverID returned by the cdidriverRegister() routine must be used to free buffers that have been allocated from the CSTA Services Driver Interface. The driver is not responsible for the memory resources allocated by the CSTA Services Driver Interface since they are allocated from the Operating System. The CSTA Services Driver Interface is responsible for freeing these memory resources. The driver is responsible for giving the memory resources back to the CSTA Services Driver Interface via the cdifreeBuffer() routine or sending the buffer to the Cserver via the cdisendToCSTA() routine.
Cserver Notes

The **driverID** must be used to free buffers that have been allocated from this CSTA Services Driver Interface. The Cserver is responsible for giving the memory resources back to the CSTA Services Driver Interface via the `cdiFreeBuffer()` routine after a message has been processed, or sending the message to the driver via the `cdiSendToDevice()` routine. The CSTA Services Driver Interface is part of the Cserver, and the memory allocated for the CSTA Services Driver Interface must be released before the Cserver can unload.
7.5. cdiSendToCSTA()

This function allows the PBX Driver to send a message buffer to the Cserver. The message will be queued until a corresponding cdiReceiveFromDriver() routine is called by the Cserver. 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 cdiAllocBuffer() routine and populated by the driver.

Syntax

```c
#include <cdi.h>

CDIReturn_t cdiSendToCSTA(
    CDIHandle_t driverID, /* INPUT */
    char *bufptr, /* INPUT */
    CDIPriority_t priority); /* INPUT */
```

Parameters

- **driverID**
  This is the value of the handle returned by the cdiDriverRegister() function call. This handle uniquely identifies the CSTA Services Driver Interface.

- **bufptr**
  This parameter is a pointer to the start of the buffer returned by the cdiAllocBuffer() function call or a cdiReceiveFromCSTA() function call. After the cdiSendToCSTA() 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, CDI_NORMAL_MESSAGE, should be used for all non-priority messages, and CDI_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 Cserver before normal messages.

Return Values

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

- **CDI_ERR_BAD_DRVRID**
  This error indicates that the driverID specified in the cdiSendToCSTA() function is not valid.
CDI_ERR_BAD_BUF

This error indicates that the memory block pointed to by bufptr is not currently allocated from the CSTA Services Driver Interface by the driver.

CDI_ERR_EINVAL

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

CDI_ERR_NOT_YOUR_BUFFER

This error indicates that the driverID specified did not match the driverID stored with this CSDI Buffer when the buffer was created via the cdiAllocBuffer() call. A driver is only allowed to send CSDI buffers that was originally allocated for this driver.

Note: All CSDI buffers allocated by the Cserver to be sent across the CSDI to a driver are allocated with that Driver's ID.

When this occurs the CSDI will place an entry in the Error Log.

CDI_ERR_ESYS

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

Comments

This function sends a message from the driver to the Cserver. The message will be queued until a corresponding cdiReceiveFromDriver() routine is called by the Cserver. 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 via the cdiAllocBuffer() routine.

Driver Notes

The PBX Driver is responsible for issuing a cdiDriverRegister() function call to specify the maximum number of bytes that can be allocated for message buffers by the Cserver or the driver for this interface. The driverID returned by the cdiDriverRegister() routine must be used to send a message to the Cserver via the cdiSendToCSTA() function call. The driver must allocate a message buffer via the cdiAllocBuffer() function call prior to calling cdiSendToCSTA(), and the driver is no longer responsible for that message buffer after cdiSendToCSTA() has completed successfully.
Cserver Notes

The Cserver is responsible for calling `cdiReceiveFromDriver()` to retrieve messages from its queues in a timely manner, and the Cserver must give the memory buffer back to the CSTA Services Driver Interface via the `cdiFreeBuffer()` routine after the message has been processed, or send this message buffer back to the driver via the `cdiSendToDriver()` routine.
7.6. cdiReceiveFromCSTA()

This function is called by the PBX Driver in order to receive a populated message buffer from the Cserver. The buffer will be owned by the driver until it is repopulated with a response message and sent back to the Cserver via the cdiSendToCSTA() routine or until it is deallocated by the driver via the cdiFreeBuffer() routine. This routine will only be able to receive a buffer if the routine cdiSendToDriver() has been previously executed by the Cserver to send a message to the driver.

Syntax

#include <cdi.h>

CDIReturn_t cdiReceiveFromCSTA(
    CDIHandle_t driverID, /* INPUT */
    char **bufptr); /* OUTPUT */

Parameters

driverID

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

bufptr

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

Return Values

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

CDI_ERR_BAD_DRV RID

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

CDI_ERR_DRV_R_UNREGISTERED

This error indicates that the driver unregistered during the time this function blocked waiting for a message from the Cserver.

CDI_ERR_ESYS

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

Comments
The *cdiReceiveFromCSTA()* function receives a message that was sent by the Cserver across the CSTA Services Driver Interface. The buffer will be owned by the driver until it is sent back to the Cserver via the *cdiSendToCSTA()* routine or until it is deallocated via the *cdiFreeBuffer()* routine. This routine will only be able to receive a buffer if the routine *cdiSendToDriver()* has been previously executed by the Cserver to send a message to the driver.

**Driver Notes**

The PBX Driver is responsible for issuing a *cdiDriverRegister()* function call to specify the maximum number of bytes that can be allocated for message buffers by the Cserver or the driver for this interface. The *driverID* returned by the *cdiDriverRegister()* routine must be used for the *cdiReceiveFromCSTA()* routine to receive messages from the Cserver over the CSTA Services Driver Interface. The driver is responsible for calling *cdiReceiveFromCSTA()* to retrieve messages from its queues in a timely manner, and the driver must give the memory buffer back to the CSTA Services Driver Interface via the *cdiFreeBuffer()* routine after the message has been processed, or the message can be sent back to the Cserver via the *cdiSendToCSTA()* routine.

**Warning**

The message buffer returned by this function should not be directly returned to the Operating System by the driver. The driver should return this buffer back to the CSDI as outlined above.
7.7. `cdiDriverSanity()`

This function is called by the PBX Driver once a minute to report to the Cserver (and Tserver) that it is alive and functioning. This function calls the Tserver `tdiDriverSanity()` function call. 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 Administrator application if it is up and running. The driver need not call this function if it reports system status at least once a minute.

Syntax

```c
#include <cdi.h>

CDIReturn_t cdiDriverSanity(
    CDIHandle_t driverID) /* INPUT */
```

Parameters

`driverID`

This is the unique identification number given to the driver when it registered with the Cserver via `cdiDriverRegister()`.

Return Values

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

- `CDI_ERR_BAD_DRVRIID` This error indicates that the `driverID` specified in the `cdiReceiveFromCSTA()` function is not valid.

Driver Notes

The PBX Driver is responsible for calling this function at least once every minute. Alternatively, the driver may send a system status message at least once every minute.
7.8. `cdiQueueSize()`

This function will return the current status of the message buffers used for the CSTA Services Driver Interface. This routine returns a count of the messages in each of the (four) possible states: queued to the Cserver, queued to the driver, "owned" by the driver, or "owned" by the Cserver.

Syntax

```c
#include <cdi.h>

CDIReturn_t cdiQueueSize(
    CDIHandle_t driverID,  /* INPUT */
    CDIQqueue_info_t *queue_descriptor); /* OUTPUT */
```

Parameters

`driverID`
This is the unique identification number given to the driver when it registered with the Cserver via the `cdiDriverRegister()` routine.

`queue_descriptor`
This parameter returns the following information in the `CDIQqueue_info_t` structure:

```c
typedef struct {
    int queued_to_driver;
    int queued_to_Cserver;
    int allocd_by_driver;
    int allocd_by_Cserver;
} CDIQqueue_info_t;
```

Where the sum of all of these fields equals the total number of messages currently allocated for this CSTA 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_Cserver`
This count specifies the number of message buffers that are currently queued to the Cserver.
**allocd_by_driver**

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

**allocd_by_Cserver**

This parameter specifies the number of message buffers that are currently allocated to the Cserver. Message buffers are allocated to the Cserver if the Cserver has performed a `cdiAllocBuffer()` or a `cdiReceiveFromDriver()` function call.

**Return Values**

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

- **CDI_ERR_BAD_DRVRID** This error indicates that the **driverID** specified in the `cdiQueueSize()` function is not valid.

**Comments**

This routine is exported from the Cserver for use by the Drivers, and it can be called internally by the Cserver. It provides a method for determining how many message buffers are queued in either direction across the CSTA Services Driver Interface. Using this function the Cserver and PBX 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. cdiMemAllocSize()

This function will return the current status of the memory used for message buffers by a Cserver-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 Cserver, queued to the driver, "owned" by the driver, or "owned" by the Cserver.

Syntax

```c
#include <cdi.h>
CDIReturn_t cdiMemAllocSize (
    CDIHandle_t driverID, /* INPUT */
    CDIMemAlloc_info_t *mem_descriptor); /* OUTPUT */
```

Parameters

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

**mem_descriptor**
This parameter returns the following information in the CDIMemAlloc_info_t structure:

```c
typedef struct {
    unsigned long bytes_queued_to_driver;
    unsigned long bytes_queued_to_Cserver;
    unsigned long bytes_allocated_by_driver;
    unsigned long bytes_allocated_by_Cserver;
}CDIMemAlloc_info_t;
```

Where the sum of these fields is the total number of bytes allocated for message buffers for this Cserver-Driver interface. The fields of CDIMemAlloc_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 does not include the (12) bytes that are part of the overhead for each message buffer allocated from the Cserver-Driver interface.

**bytes_queued_to_Cserver**
This parameter specifies the number of bytes in message buffers that are currently queued to the Cserver. This count does not include the (12) bytes that are part of the overhead for each message buffer allocated from the Cserver-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 has performed a `cdiAllocBuffer()` or a `cdiReceiveFromCSTA()` function call. This count does not include the (12) bytes that are part of the overhead for each message buffer allocated from the Cserver-Driver interface.

**bytes_allocd_by_Cserver**

This parameter specifies the number of bytes in message buffers that are currently allocated to the Cserver. Message buffers are allocated to the Cserver if the Cserver has performed a `cdiAllocBuffer()` or a `cdiReceiveFromDriver()` function call. This count does not include the (12) bytes that are part of the overhead for each message buffer allocated from the Cserver-Driver interface.

**Return Values**

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

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

**Comments**

This routine is exported from the Cserver for use by the Drivers, and it can be called internally by the Cserver. It provides a method for determining how memory is used for message buffers in the Cserver-Driver Interface. Using this function the Cserver and PBX 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. cdiGetCDISize ( )

This function will return the current size setting of the CSTA Services Driver Interface. This routine returns the maximum memory utilization and the hi-water and lo-water mark values defined for this CSDI interface.

Syntax

```c
#include <cdi.h>

CDIReturn_t cdiGetCDISize (
    CDIHandle_t driverID, /* INPUT */
    CDIBuf_info_t *buffer_descriptor); /* OUTPUT */
```

Parameters

**driverID**

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

**buffer_descriptor**

This is a pointer to a buffer descriptor structure that will be filled in with the information about the memory that can be allocated for this CSTA Services Driver Interface. This parameter may not be set to NULL.

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

Return Values

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

**CDI_ERR_BAD_DRVRID** This error indicates that the `driverID` specified in the `cdiQueueSize()` function is not valid.
Comments

This routine is exported from the Cserver for use by the Drivers, and it can be called internally by the Cserver. It provides a method for determining how many message buffers are queued in either direction across the CSTA Services Driver Interface. Using this function the Cserver and PBX 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. Note that presently no mechanism exists to change the memory utilization parameters returned by this interface, however, the equivalent parameters in the TSDI can be changed via the OA&M interface to the Tserver.
7.11. `cdiGetCSTAVersion()`

This function will get the customer and internal version of the Cserver. This function also returns a list of CSDI versions and CSTA protocol versions that the Cserver supports.

**Syntax**

```c
#include <cdi.h>
CDIReturn_t cdiGetCSTAVersion(
    char *customer_version, /* OUTPUT */
    char *internal_version, /* OUTPUT */
    CDIVersion_t *csdi_versions, /* OUTPUT */
    CDIProtocol_t *protocol_versions) /* OUTPUT */
```

**Parameters**

* `customer_version`
  This is the customer version of the CSTA Server.

* `internal_version`
  This is the internal version of the CSTA Server.

* `csdi_versions`
  This is a bit field indicating the versions of the CSDI that the CSTA Server supports. Each bit corresponds to a particular CSDI version. Currently, only one CSDI version exists. A value of `CDI_CSDI_VERSION_1` (1) indicates the first version of the CSDI.

* `protocol_versions`
  This is a bit field indicating the CSTA protocol versions that the CSTA Server supports. Each bit corresponds to a particular CSTA protocol version. Currently, only one CSTA protocol version is supported. A value of `CDI_PROTOCOL_VERSION_1` (1) indicates the June 1992 version of ECMA-180.

**Return Values**

This function always returns `CDI_SUCCESS`. 
Comments

A value of NULL may be passed in any parameter to cdiGetCSTAVersion() if the associated information is not of interest to the caller. The maximum length of a returned customer_version or internal_version string is CDI_MAX_VERSION_STRING, not including the terminating NULL.
8. CSDI, ACS, and CSTA Message Interface Header Files

8.1. cdi.h
All header files needed for CSDI development are contained on the CSTA SDK disk which is part of the DRV-SDK kit.

8.2. tdi.h
All header files needed for CSDI development are contained on the CSTA SDK disk which is part of the DRV-SDK kit.

8.3. acs.h
All header files needed for CSDI development are contained on the CSTA SDK disk which is part of the DRV-SDK kit.

8.4. acsdefs.h
All header files needed for CSDI development are contained on the CSTA SDK disk which is part of the DRV-SDK kit.

8.5. csta.h
All header files needed for CSDI development are contained on the CSTA SDK disk which is part of the DRV-SDK kit.

8.6. cstadefs.h
All header files needed for CSDI development are contained on the CSTA SDK disk which is part of the DRV-SDK kit.

9. References


[TSDI] NetWare Telephony Services: PBX Driver Interface Specification Releases 1.0 and 2.0.


[X.208] Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1), The International Telegraph and Telephone Consultative Committee (CCITT)

[X.209] Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), The International Telegraph and Telephone Consultative Committee (CCITT)
Recommendation X.219: Remote Operations: Model, Notation and Service Definition, The International Telegraph and Telephone Consultative Committee (CCITT)


