Avaya MultiVantage®
Application Enablement Services
TSAPI for Avaya Communication Manager
Programmer's Reference

02-300544
Release 4.2
May 2008
Issue 4

Notice

While reasonable efforts were made to ensure that the information in this document was complete and accurate at the time of printing, Avaya Inc. can assume no liability for any errors. Changes and corrections to the information in this document might be incorporated in future releases.

Documentation disclaimer

Avaya Inc. is not responsible for any modifications, additions, or deletions to the original published version of this documentation unless such modifications, additions, or deletions were performed by Avaya. Customer and/or End User agree to indemnify and hold harmless Avaya, Avaya's agents, servants and employees against all claims, lawsuits, demands and judgments arising out of, or in connection with, subsequent modifications, additions or deletions to this documentation to the extent made by the Customer or End User.

Link disclaimer

Avaya Inc. is not responsible for the contents or reliability of any linked Web sites referenced elsewhere within this documentation, and Avaya does not necessarily endorse the products, services, or information described or offered within them. We cannot guarantee that these links will work all the time and we have no control over the availability of the linked pages.

Warranty

Avaya Inc. provides a limited warranty on this product. Refer to your sales agreement to establish the terms of the limited warranty. In addition, Avaya's standard warranty language, as well as information regarding support for this product, while under warranty, is available through the Avaya Support Web site:

http://www.avaya.com/support

License

USE OR INSTALLATION OF THE PRODUCT INDICATES THE END USER'S ACCEPTANCE OF THE TERMS SET FORTH HEREIN AND THE GENERAL LICENSE TERMS AVAILABLE ON THE AVAYA WEB SITE http://support.avaya.com/LicenseInfo/ ("GENERAL LICENSE TERMS"). IF YOU DO NOT WISH TO BE BOUND BY THESE TERMS, YOU MUST RETURN THE PRODUCT(S) TO THE POINT OF PURCHASE WITHIN TEN (10) DAYS OF DELIVERY FOR A REFUND OR CREDIT.

Avaya grants End User a license within the scope of the license types described below. The applicable number of licenses and units of capacity for which the license is granted will be one (1), unless a different number of licenses or units of capacity is specified in the Documentation or other materials available to End User. "Designated Processor" means a single stand-alone computing device. "Server" means a Designated Processor that hosts a software application to be accessed by multiple users. "Software" means the computer programs in object code, originally licensed by Avaya and ultimately utilized by End User, whether as stand-alone Products or pre-installed on Hardware. "Hardware" means the standard hardware Products, originally sold by Avaya and ultimately utilized by End User.

License type(s)

Designated System(s) License (DS). End User may install and use each copy of the Software on only one Designated Processor, unless a different number of Designated Processors is indicated in the Documentation or other materials available to End User. Avaya may require the Designated Processor(s) to be identified by type, serial number, feature key, location or other specific designation, or to be provided by End User to Avaya through electronic means established by Avaya specifically for this purpose.

Concurrent User License (CU). End User may install and use the Software on multiple Designated Processors or one or more Servers, so long as only the licensed number of Units are accessing and using the Software at any given time. A "Unit" means the unit on which Avaya, at its sole discretion, bases the pricing of its licenses and can be, without limitation, an agent, port or user, an e-mail or voice mail account in the name of a person or corporate function (e.g., webmaster or helpdesk), or a directory entry in the administrative database utilized by the Product that permits one user to interface with the Software. Units may be linked to a specific, identified Server.

Copyright

Except where expressly stated otherwise, the Product is protected by copyright and other laws respecting proprietary rights. Unauthorized reproduction, transfer, and or use can be a criminal, as well as a civil, offense under the applicable law.

Third-party components

Certain software programs or portions thereof included in the Product may contain software distributed under third party agreements ("Third Party Components"), which may contain terms that expand or limit rights to use certain portions of the Product ("Third Party Terms"). Information identifying Third Party Components and the Third Party Terms that apply to them is available on the Avaya Support Web site:

http://support.avaya.com/ThirdPartyLicense/

Preventing toll fraud

"Toll fraud" is the unauthorized use of your telecommunications system by an unauthorized party (for example, a person who is not a corporate employee, agent, subcontractor, or is not working on your company's behalf). Be aware that there can be a risk of toll fraud associated with your system and that, if toll fraud occurs, it can result in substantial additional charges for your telecommunications services.

Avaya fraud intervention

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

http://www.avaya.com/support

Trademarks

Avaya and the Avaya logo are either registered trademarks or trademarks of Avaya Inc. in the United States of America and/or other jurisdictions.

CallVisor is a registered trademark of Avaya Inc.

DEFINITY is a registered trademark of Avaya Inc.

MultiVantage is a registered trademark of Avaya Inc.

All other trademarks are the property of their respective owners.

Downloading documents

For the most current versions of documentation, see the Avaya Support Web site:

http://www.avaya.com/support

Avaya support

Avaya provides a telephone number for you to use to report problems or to ask questions about your product. The support telephone number is 1-800-242-2121 in the United States. For additional support telephone numbers, see the Avaya Support Web site:

http://www.avaya.com/support

Contents

About this document

This document, the *Avaya MultiVantage Application Enablement Services TSAPI for Communication Manager Programmer Reference* is the primary documentation resource for developing and maintaining TSAPI based applications in an Avaya Communication Manager environment. TSAPI is the acronym for Telephony Services Application Programming Interface.

Intended audience

This programming guide is intended for C programmers (C or C++) who have a working knowledge of the following:

- Ecma International Standards for Computer Supported Telecommunications Applications (ECMA-179 and ECMA-180)
- Telephony Services Application Programming Interface (TSAPI) Specification. This is documented by the *Avaya MultiVantage Application Enablement Services TSAPI Programmer Reference*, 02-300545
- Telecommunications applications

Structure and organization of this document

Use this chapter summary to become familiar with the structure and contents of this document.

- Chapter 1: Overview of the TSAPI Client and the TSAPI SDK on page 27 provides a brief overview of the AE Services TSAPI Service.
- Chapter 2: The TSAPI Programming Environment on page 31 describes the tools that are provided with the TSAPI SDK. This chapter also provides some basic programming tips and some advanced programming tips.
- Chapter 3: Control Services on page 61 describes the control services that are provided by Telephony Services Application Programming Interface (TSAPI). This chapter is based on the “Control Services” chapter in the *Avaya MultiVantage Application Enablement Services TSAPI Programmer Reference*, 02-300545. This information applies at the TSAPI interface level, and it is not specific to Communication Manager. You will need to use these control services in a Communication Manager environment. To avoid having you refer to the TSAPI Programmer Reference for information about control services, this document includes the information.
- Chapter 4: CSTA Service Groups supported by the TSAPI Service on page 137 describes the CSTA Service groups that the TSAPI Service supports.
- Chapter 5: Avaya TSAPI Service Private Data on page 161 describes the private data services provided by the TSAPI Service. This chapter also includes information about how to manage private data using the private data version control mechanism.
- Chapter 6: Call Control Service Group on page 185 describes the group of services that enable a telephony client application to control a call or connection on Communication Manager. These services are typically used for placing calls from a device and controlling any connection on a single call as the call moves through Communication Manager.
- Chapter 7: Set Feature Service Group on page 337 describes the services that allow a client application to set switch-controlled features or values on a Communication Manager device.
- Chapter 8: Query Service Group on page 367 describes the services that allow a client application to query the switch to provide the state of device features and static attributes of a device.
- Chapter 9: Snapshot Service Group on page 431 describes the services that enable the client to “take a snapshot” of information about a particular call and information concerning calls associated with a particular device.
- Chapter 10: Monitor Service Group on page 443 describes the three types of monitor services the TSAPI Service provides for Communication Manager.
- Chapter 11: Event Report Service Group on page 491 describes event messages (or reports) from Communication Manager to the TSAPI Service.

- Chapter 12: Routing Service Group on page 695 describes the services that allow the switch to request and receive routing instructions for a call.
- Chapter 13: System Status Service Group on page 751 describes the services that allow an application to receive reports on the status of the switching system.
- Universal Failure Events on page 785, describes ACS Universal Failure Events.
- Appendix B: Summary of Private data support on page 811 describes previous private data versions of AE Services.
- Appendix C: Server-Side Capacities on page 835 describes server-side capacities, which include Avaya Communication Manager capacities and AE Services TSAPI Service capacities.

Reason for Reissue

This section highlights changes to the TSAPI for Communication Manager Programmer's Reference for AE Services Releases 4.0 through 4.2:

General maintenance updates

The AE Services 4.2 TSAPI for Communication Manager Programmer's reference has been reissued to make minor corrections to manual pages.

Private Data Version 8

AE Services 4.1 introduces private data version 8, which includes the following features.

- Single Step Transfer Call - see Single Step Transfer Call (Private Data Version 8 and later) on page 327.
- Calling Device in Failed Event - see Calling Device in Failed Event on page 168.
- New Get API Capabilities confirmation event - see CSTA Get API Capabilities confirmation structures for Private Data Version 8 on page 171
- A new private data parameter, flowPredictiveCallEvents, has been added to the CSTAMonitorCallsViaDevice service. For more information, see Monitor Calls Via Device Service on page 462.

Route Registration Request service update

Beginning with AE Services 4.0, The TSAPI Service will allow an application to re-establish a route registration request due to a service interruption, such as a network outage between the client and the AE Server. For information about functional changes, see the following sections of this document

- Routing transactions on page 53
- Route Register Service on page 710
- Route Register Abort Event on page 705

TSAPI client connections over secure links

Beginning with AE Services 4.1.0, the TSAPI service provides the option for configuring secure application links between the TSAPI client and the AE Services server.

In terms of the TSAPI client you will need to set up the configuration file (tslib.ini, for Windows or tslibrc, for Linux) to select the AE Services Server (AE Server) which is configured for secure TLINKs (described next in Server Implementation notes for TSAPI client connections). To establish a session, TSAPI based applications use the acsOpenStream( ) service to open a TLINK. As a result of accommodating secure links, acsOpenStream() provides several new return values. For more information, see acsOpenStream ( ) on page 73.

Server Implementation notes for TSAPI client connections: To implement client connections over secure TLINKs, you need to administer the AE Services Server, using the Application Enablement Services (AE Services) Operations, Administration, and Maintenance (OAM) Web-based interface, as follows:

- (Optional) Administer "Encrypted TLINK Ports" on the Ports OAM page.To access this setting in OAM, select **CTI OAM Administration** > **Administration** > **Network Configuration** > **Ports**.
- Administer TSAPI links with the **Encrypted** security setting on the Add / Edit TSAPI Links OAM page. To access This page in OAM, select **CTI OAM Administration** > **Administration** > **CTI Link Admin** > **TSAPI Links**. From the TSAPI Links OAM page, select **Add Link** or **Edit Link.**

For more information, see the *Avaya MultiVantage Application Enablement Services Administration and Maintenance Guide*, 02-300357 and the AE Services OAM Help pages.

TSAPI client heartbeat

The AE Services TSAPI Release 4.1.0 client automatically provides a client heartbeat. For more information, see the following topics:

- Opening an ACS stream on page 64
- acsOpenStream ( ) on page 73.
- acsSetHeartbeatInterval() on page 107

Alternate TLINK capability

Beginning with AE Services 4.1.0, the TSAPI Service provides TSAPI-based applications with the ability to specify an optional list of alternate TLINKs automatically and transparently. The alternate TLINKs are only used if the TLINK specified in the open stream call was not available at the time that procedure was executed.

When multiple AE Servers are used as alternates, the username and password specified by the application in the acsOpenStream() request should be configured identically for each AE Server.

For more information, see acsGetServerID() on page 103.

AE Services 4.1.0 clients and backward compatibility

Application Enablement Services is the software platform for the TSAPI Service (Tserver).

AE Services 4.1.0 supports the AE Services TSAPI 4.1.0 client and is backward compatible with Release 4.0.x, AE Services TSAPI client Release 3.1.x and Avaya Computer Telephony R1.3.

Note:
The AE Services 4.1.0 enhancements are not supported with any of the supported prior releases of the AE Services server -- AE Services Release 4.0 or AE Services 3.1.x . You can use an AE Services Server (3.1.x or 4.0.0) with a the TSAPI Release 4.1.0 client, but the application will not have access to the new AE Services 4.1.0 features.

About installing the SDK

This programmer reference assumes that you have installed the AE Services TSAPI client and the Software Development Kit (SDK). The *Avaya MultiVantage Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide*, 02-300543, provides instructions for installing the TSAPI client and the SDK in Chapter 1, “Installing AE Services TSAPI clients and SDKs.”

When you install the SDK software, you install the TSAPI SDK header files, libraries, samples and tools. If you install the TSAPI Client only, you will not have access to these SDK components.

In terms of working with the software, your starting point in this programmer reference is Chapter 2: The TSAPI Programming Environment. Chapter 2 describes the names and locations of the SDK components, and provides some basic information about using them.

Related Documents

This section provides references for documents that serve as the basis for this document as well as documents that contain additional information about AE Services and Communication Manager features.

- Related Ecma International documents on page 19
- Related Avaya documents on page 20

Related Ecma International documents

This programming reference is based on the following Ecma International documents:

- ECMA -179, “Services for Computer Supported Telecommunications Applications (CSTA) Phase I, “defines the relationship between an application and a switch. It also defines the CSTA Services that an application can request.
- ECMA -180, “Protocol for Computer Supported Telecommunications Applications (CSTA) Phase I,” defines a Protocol for Computer-Supported Telecommunications Applications (CSTA) for OSI Layer 7 communication between a computing network and a telecommunications network. ECMA-180 specifies application protocol data units (APDUs) for the services described in ECMA-179.
- ECMA - 269, “Services for Computer Supported Telecommunications Applications (CSTA) Phase III,” Phase III of CSTA extends the previous Phase I and Phase II Standards.

Related Avaya documents

You can find additional information in the following documents.

- *Avaya MultiVantage Application Enablement Services TSAPI Programmer Reference*, 02-300545 (also referred to as the TSAPI Specification) The *Avaya MultiVantage Application Enablement Services TSAPI Programmer Reference* is the generic description of TSAPI as a standard programming interface. Use the TSAPI Specification if you need to brush-up on your TSAPI skills or refresh your knowledge of TSAPI.

  Use the document you are currently reading.as your primary reference for developing and maintaining TSAPI applications. It describes how to program to the TSAPI interface in an Avaya Communication Manager environment.

- *Avaya MultiVantage Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide*, 02-300543.

  Use this document to install the software development kits (SDKs) that this document, the TSAPI for MultiVantage Programmer's Reference. describes.

- *Avaya MultiVantage Application Enablement Services Administration and Maintenance Guide,* 02- 300357. Use this administration guide for information about the configuration and operation of the AE Services TSAPI Service.

- *AE Services OAM Help* (included with the AE Services operations, administration, and maintenance (OAM) interface). Use this on-line reference as a supplement to the administration guide for information about the configuration and operation of the AE Services TSAPI Service.

- *Administrator Guide for Avaya Communication Manager*, 03-300509

  Use this administrator guide when you need information about switch setup and operation.

Web based training

The Avaya Developer Connection program (DevConnect) provides a series of Web based training modules called "Avaya Application Enablement Services." If you are interested in developing TSAPI applications, DevConnect provides a training module that teaches you how to develop applications using Telephony Services Application Programming Interface (TSAPI).

- Log in to DevConnect (devconnect.avaya.com).
- From the Welcome page, select **Avaya Application Enablement Services - In-Depth Technical Training**, and follow the links to get to the DevConnect Training site.

Note:
To access Web based training, you must be a registered member of DevConnect.

Customer Support

For questions about Application Enablement Services, TSAPI Service operation, call 1-800-344-9670.

Conventions used in this document

This document uses the following conventions.

Convention	Example	Usage
plain monospace	`include <acs.h>`	Coding examples. **Note:** Coding examples contain operators and special characters that are part of the C programming syntax. For example, the angle brackets in the example are part of C syntax.
bold	**Start**	In text descriptions, bold can indicate the following. • Mouse and keyboard selections • function calls • command names • field names (field names refer to alphanumeric text you would type in a text box or a selection you would make from a drop-down list. • special emphasis
uppercase • ACS • CSTA	**ACSUniversalFailureConfEvent** **CSTAGetDeviceConfEvent**	When the terms **ACS** and **CSTA** are uppercased, they refer to structures.
lowercase • acs • csta	**acsOpenStream** **cstaQueryCallMonitor**	When the terms **acs** and **csta** are lowercased, they refer to function calls.

Format of Service Description Pages

Chapters 3 through 13 of this document contain service descriptions. Table 1 describes the general format and content of the service descriptions.

Table 1: Service Description page elements

Element	Description
Summary	Short description of the service in a list format.
Direction	Direction of the service request or event report across the TSAPI interface: ● Client to Switch client/application to switch/TSAPI Service ● Switch to Client switch/TSAPI Service to client/application
Function and Confirmation Event:	CSTA service request function and CSTA confirmation event as defined in the *Avaya MultiVantage Application Enablement Services TSAPI Programmer Reference*, 02-300545
Private Data Function and Private Data Confirmation Event	Private data setup function and private data confirmation event, if any. This function may be called to setup private parameters, if any. This function returns an error, if there is an error in the private parameters. An application should check the return value to make sure that the private data is set up correctly before sending the request to the TSAPI Service.
Service Parameters:	List of parameters for this service request. Common ACS parameters such as acsHandle, invokeID, and privateData are not shown.
Private Parameters:	List of parameters that can be specified in private data for this service request.
Ack Parameters:	List of parameters in the confirmation event for the positive acknowledgment from the server. Common ACS parameters such as acsHandle, eventClass, eventType, and privateData are not shown.
Ack Private Parameters:	List of parameters in the private data of the confirmation event for the positive acknowledgment from the server.
Nak Parameter: ***universalFailure***	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain one of the error values described in the CSTAUniversalFailureConfEvent on page 160.
Functional Description	Detailed description of the telephony function that this CSTA Service provides in a TSAPI Service CSTA environment.
Service Parameters	Indicates the parameter type.
parameter	Detailed information for each parameter in the service request. A noData indicator means that it requires no additional parameters other than the common ACS parameters.
mandatory	[mandatory] This parameter is mandatory as defined in Standard ECMA-179. It must be present in the service request. If not, the service request will be denied with OBJECT_NOT_KNOWN.

Table 1: Service Description page elements (continued)

Element	Description
mandatoryPartially	[mandatory - partially supported] This parameter is mandatory as defined in Standard ECMA-179. However, the TSAPI Service CSTA can only support part of the parameter due to Communication Manager feature limitations. The TSAPI Service sets a Communication Manager default value for the portion not supported.
mandatoryNotSupt	[mandatory - not supported] This parameter is mandatory as defined in Standard ECMA-179. However, The TSAPI Service CSTA does not support this parameter due to Communication Manager feature limitations. “Not supported” means that whether the application passes it or not, the value specified will be ignored and a default value will be assigned. If this is a parameter (for example, event report parameter) returned from the switch, the TSAPI Service sets a Communication Manager default value for this parameter.
optional	[optional] This parameter is optional as defined in Standard ECMA-179. It may or may not be present in the service request. If not, the TSAPI Service sets a Communication Manager default value.
optionalSupported	[optional - supported] This parameter is optional as defined in Standard ECMA-179, but it is always supported.
optionalPartially	[optional - partially supported] This parameter is optional as defined in Standard ECMA-179. However, the TSAPI Service CSTA Services can only support part of the parameter due to Communication Manager feature limitations. The part that is not supported will be ignored, if it is present.
optionalNotSupport	[optional - not supported] This parameter is optional as defined in Standard ECMA-179, but it is not supported by The TSAPI Service CSTA Services. “Not supported” means that whether the application passes it or not, the value specified will be ignored and the TSAPI Service will assign a Communication Manager default value.
optionalLimitedSupt	[optional - limited support] This parameter is optional as defined in Standard ECMA-179, but it is not fully supported by the TSAPI Service CSTA Services. An application must understand the limitations of this parameter in order to use the information correctly. The limitations are described in the Detailed Information section associated with each service.
Private Service Parameters:	
parameter	Detailed information for each private parameter in the service request.
mandatory	[mandatory] This parameter is mandatory for the specific service. It must be present in the private data of the request. If not, the service request will be denied by the TSAPI Service with OBJECT_NOT_KNOWN.
optional	[optional] This parameter is optional for the specific service. It may or may not be present in the private data. If not, TSAPI Service will assign a Communication Manager default value.
optionalNotSupported	[optional - not supported] This parameter is optional for the specific service. This parameter is reserved for future use. It is ignored for the current implementation.
Ack Parameters:	
parameter	Detailed information for each parameter in the service confirmation event. A noData indicator means that the TSAPI Service sends no additional parameters other than the confirmation event itself along with the common ACS parameters.

Table 1: Service Description page elements (continued)

Element	Description
Ack Private Parameters:	
parameter	Detailed information for each parameter in the private data of the service confirmation event.
Nak Parameter:	
universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain one of the error values described in the CSTAUniversalFailureConfEvent on page 160.
Detailed Information:	Detailed information about switch operations, feature interactions, restrictions, and special rules.
Syntax:	C-declarations of the TSAPI function and the confirmation event for this service. See Common ACS Parameter Syntax on page 25.
Private Parameter Syntax:	C-declarations of the private parameters and the set up functions and of the private parameters in the confirmation event for this service.
Example:	Programming examples are given for some of the services and events.

Common ACS Parameter Syntax

Here an example of the common ACS parameter syntax used on the service description pages.

```
typedef  unsigned  long InvokeID_t;

typedef unsigned  short ACSHandle_t;

typedef unsigned  short EventClass_t;

typedef unsigned  short EventType_t;

// defines for ACS event classes

#define     ACSREQUEST              0
#define     ACSUNSOLICITED          1
#define     ACSCONFIRMATION         2

// defines for CSTA event classes

#define     CSTAREQUEST             3
#define     CSTAUNSOLICITED         4
#define     CSTACONFIRMATION        5

#define CSTAEVENTREPORT 6
```

Chapter 1: Overview of the TSAPI Client and the TSAPI SDK

This chapter provides a brief history of TSAPI (Telephony Services Application Programming Interface). It includes the following topics.

Introduction

Application Enablement Services (AE Services) TSAPI for Communication Manager is a library interface that is designed exclusively for use with Avaya Communication Manager. It is a standards based library based on Ecma International Standards and the Telephony Services Application Programming Interface (TSAPI) Specification. This historical summary describes the relationship between the Ecma International standards, the TSAPI Specification, and TSAPI for Communication Manger. The following topics describe how these pieces fit together at a conceptual level.

Ecma International and the CSTA Standards

Ecma International is an international standards organization. Two Ecma standards, ECMA-179 and ECMA-180 are about Computer-Supported Telecommunications Applications (CSTA), and they are often referred to as “CSTA” documents. These two CSTA documents form the basis for Computer Telephony Integration (CTI).

- ECMA-179 defines the relationship between an application and a switch. It also defines the CSTA Services that an application can request.
- ECMA-180 defines a Protocol for Computer-Supported Telecommunications Applications (CSTA) for OSI Layer 7 communication between a computing network and a telecommunications network. ECMA-180 specifies application protocol data units (APDUs) for the services described in ECMA-179.

For more information about Ecma International and to get the standards go to the Ecma International Web Site:

http://www.ecma-international.org/memento/index.html

The TSAPI Specification

The Telephony Services Application Programming Interface (TSAPI) specification is an implementation of the ECMA-179 and ECMA-180 standards. It is a generic, switch-independent API that describes how to implement Computer Telephony Integration (CTI) in a switch-independent way. This generic specification is described in the *Application Enablement Services TSAPI Programmer’s Reference,* 02-300545. It describes TSAPI at the TSAPI interface level, and forms the basis for this document, the *Avaya Application Enablement Services TSAPI for Communication Manager Programmer Reference*.

TSAPI for Avaya Communication Manager

TSAPI for Avaya Communication Manager is an implementation of the generic TSAPI Specification. Stated another way, TSAPI for Avaya Communication Manager is a switch-specific API that provides the C programming community (C and C++ programmers) with a way to implement CTI in Avaya Communication Manager environment.

This document, the AE Services TSAPI for Communication Manager is your primary documentation resource for developing and maintaining TSAPI applications. It describes the CSTA services that are available for interacting with Avaya Communication Manager.

The TSAPI Client

The TSAPI Client provides applications with access to Avaya Communication Manager call processing. The primary component of the TSAPI Client is the TSAPI library. The TSAPI library is the C library of function calls that enables an application to request CSTA Services. Additionally the TSAPI client provides access to Avaya Private data. Avaya Private Data provides access to specialized features of Avaya Communication Manager. For more information about TSAPI client libraries, see Chapter 2: The TSAPI Programming Environment.

The TSAPI SDK

The TSAPI SDK provides you with the necessary tools for developing and designing a TSAPI application in a Communication Manager environment. The TSAPI SDK includes the TSAPI Client. For more information about the TSAPI SDK, see Contents of the TSAPI SDK on page 32.

Chapter 2: The TSAPI Programming Environment

The TSAPI Software Development Kit (SDK) is intended for programmers who are developing Computer Telephony Integration (CTI) applications. This chapter provides some basic information about the TSAPI programming environment. It includes the following topics.

Tip:
AE Services provides a self-paced, Web-based training module that teaches you how to develop TSAPI applications. For more information see Web based training on page 21.

Contents of the TSAPI SDK

The AE Services TSAPI SDK consists of the following components.

- headers and libraries
- TSAPI Client. The TSAPI client must be installed separately. For information about installing the TSAPI SDK and the TSAPI Client, see the *Avaya Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide*, 02-300543.
- sample code
- the TSAPI Exerciser (for Windows based clients only)

TSAPI SDK header files

The TSAPI SDK header files contain coding structures you need to use for designing and maintaining your applications. If you plan to design or update an application for compliance with Private Data you will need to use “attpriv.h” and “attpdefs.h.” For more information about private data, see Using the private data header files on page 180.

TSAPI Service client libraries

The TSAPI Service client library provides a set of functions that acts as an interface between client applications and the TSAPI Service. Applications use these functions to establish an authorized connection with the TSAPI Service and to send telephony control messages (CSTA messages) to Avaya Communication Manager. The TSAPI SDK library files are import libraries.

Table 2: TSAPI Service client libraries

Library name	Operating system	Description
CSTA32.DLL	Windows	Contain TSAPI functions (CSTA*ServiceName*) and the ACSService name services.
libcsta.so	Unix-based	
ATTPRIV32.DLL	Windows	Contain private data encoding and functions (ATT*ServiceName*) .
libattpriv.so	Unix-based	

TSAPI client library configuration file (TSLIB)

The TSAPI for Communication Manager client libraries use the TSLIB configuration file to identify the network address of the AE Services Server running the TSAPI Service.

- For Windows based clients, the configuration file is TSLIB.INI
- For Linux based clients, the configuration file is tslibrc.

See the *Avaya Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide*, 02-300543, for information about setting up the TSLIB configuration file.

Code Samples (Windows client only)

The Samples directory contains samples of complete applications that demonstrate how to program to TSAPI.

Table 3 provides a brief description of each of the TSAPI code samples. Each sample is a complete application that demonstrates how to program to TSAPI. Notice that each sample builds on the next, with the successive one implementing more TSAPI functionality than the previous one. See Table 4 and Table 5 for a list of the files that the sample applications use.

Note:
Porting this code to other platforms will require modifications to event notification.

Table 3: TSAPI Code Samples

Sample	Functionality	Summary
1. TSAPIOUT	TSAPI outgoing call handling	One device, one call ● Shows basic outgoing call handling for a single device and a single call with no redirection, conferencing, transferring, and so on. ● It includes making a call and hanging up a call.
2. TSAPIIN	TSAPI incoming call handling	One device, one call ● Adds incoming call handling to Sample 1 (no redirection, conferencing, transferring, and so on). ● It demonstrates the difference between incoming calls and outgoing calls.
3. TSAPIMUL	TSAPI multiple call handling	One device, many calls ● Adds multiple handling to Sample 2. ● Demonstrates how to keep track of multiple calls on the same device. ● Includes holding calls, retrieving calls, and redirecting calls.
4. TSAPICNF	TSAPI conference call handling	One device, many calls ● Adds conferencing and transferring to Sample 3. ● Includes tracking of many connections on a single call.

Table 4: TSAPI Sample code -- common files

File Name	Description
• TSAPI. CPP • TSAPI.H	• Interface to the TSAPI Service • Helper classes for tracking devices and calls. Includes routines for retrieving events from the CSTA32.DLL
• OPENTSRV.CPP • OPENTSRV.H	• Implementation file that handles the Open Tserver dialog • Supports the Open Tserver dialog. Authorizes the user, opens the TSAPI stream and registers the selected device with the TSAPI helper classes.
• SAMPLDLG.CPP • SAMPLDLG.H	• implementation file • Supports the main application dialog. All call related control is here: making calls, answering calls, call event handling, and so forth.
• STDAFX.CPP • STDAFX.H	• source file that includes just the standard includes • MFC files that do not contain any interesting code for the purpose of TSAPI-code demonstration
• RESOURCE.H	• Resource IDs for the application

Table 5: TSAPI Sample code -- application specific function files

Name	Description
• TSAPIOUT.CPP • TSAPIOUT.H • TSAPIOUT.RC	• Defines the class behaviors for Sample 1, the TSAPIOUT application • Main header file for the TSAPIOUT application • Initialization and resources for Sample 1.
• TSAPIIN.CPP • TSAPIIN.H • TSAPIIN.RC	• Defines the class behaviors for Sample 2, the TSAPIIN application • Main header file for the TSAPIOUT application • Initialization and resources for Sample 2.
• TSAPIMUL.CPP • TSAPIMUL.H • TSAPIMUL.RC	• Defines the class behaviors for Sample 3, the TSAPIMUL application • Main header file for the TSAPIOUT application • Initialization and resources for Sample 3.
• TSAPICNF.CPP • TSAPICNF.H • TSAPICNF.RC	• Defines the class behaviors for Sample 4, the TSAPCNF application • Main header file for the TSAPIOUT application • Initialization and resources for Sample 4.

TSAPI for Windows SDK Overview

Read this section for information about developing TSAPI applications in a Windows environment. You do not need to be familiar with the CSTA call model or API, but you should read Chapter 3: Control Services.

File locations

For information about installed files see Appendix A of the *Avaya MultiVantage Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide*, 02-300543.

Development Platforms

AE Services requires that you use Microsoft Visual C++ 6.0 or Microsoft Visual C++ 2005 for developing Windows .EXE applications. Using another compiler may require you to modify the header files, for example, to account for differences in structure alignment, size of enumerated data types, and so forth. The Win32 TSAPI library assumes the default 8-byte structure packing and an enum size of 4 bytes.

Linking to the TSAPI Library

The TSAPI for Win32 is implemented as a dynamic link library, CSTA32.DLL. Specify the CSTA32.LIB import library when compiling your application.

Note:
Applications using private data should also specify the ATTPRIV32.LIB import library.

Using Application Control Services

This section describes how to use application control services (ACS) to retrieve events on Win32 platforms.

Note:
If you are porting code that uses telephony services, you should read this section to get an overview of the differences between Win32 and other platforms.

Event Notification:

- acsEventNotify() enables asynchronous notification of incoming events via Windows messages.
- acsSetESR() enables asynchronous notification of incoming events via an application-defined callback routine. This routine will be called in the context of a background thread created by the TSAPI Library, not a thread created by the application. The callback should not invoke TSAPI Library functions.

Receiving Events

This section describes event reception using acsGetEventPoll() and acsGetEventBlock() on Win32.

Blocking Versus Polling:

acsGetEventBlock() suspends the calling thread until it receives an event. acsGetEventPoll() returns control immediately if no event is available, allowing the application to query other input sources or events.

Tip:

Calling acsGetEventPoll() repetitively can unduly consume processor time and resources, to the detriment of other applications. Instead of polling, consider creating a separate thread which calls acsGetEventBlock(), or use acsEventNotify() to receive asynchronous notifications.

Receiving Events From Any Stream:

An application may specify a NULL stream handle when calling acsGetEventPoll() or acsGetEventBlock() to request that the TSAPI Service library return the first event available on any of that application's streams.

Sharing ACS Streams Between Threads:

The ACS handle value is global to all threads in a given application process. This handle can be accessed in any thread, even threads that did not originally open the handle. For example, one thread can call the acsOpenStream() function, which returns an ACS handle. A different thread in the same process can make other TSAPI calls with the returned ACS handle. No special action is required to enable the second thread to use the handle; it just needs to obtain the handle value.

While permitted, it normally does not make sense for more than one thread to retrieve events from a single stream. The TSAPI Library allows calls from different threads to be safely interleaved, but coordination of the resulting actions and events is the responsibility of the application.

Message Trace:

The TSAPI Spy (TSSPY32.EXE) program may be used to obtain a trace of messages flowing between applications and the TSAPI Service.

Sample Code

The following Linux pseudo-code illustrates the use of the acsGetFile() function to set up an asynchronous event handler.

```
int EventIsPending = 0;

/* handleEvent() called when SIGIO is received */

void
handleEvent (int sig)
{
      EventIsPending++;
}

void
main (void)
{
      ACSHandle_tacsHandle;
      int     acs_fd;
          .
          .
          .

      /* install the signal handler */
      signal (SIGIO, handleEvent);

      /* open an ACS stream */
      acsOpenStream (&acsHandle, ...etc... );

      /* get its file descriptor */
      acs_fd = acsGetFile (acsHandle);

      /* Indicate that this process should receive */
      /* notification of pending input.
      /*
fcntl(acs_fd, F_SETOWN, getpid());

/*
 * Enable asynchronous notification of
 * pending I/O requests.
 */
fcntl(acs_fd, F_SETFL, FASYNC);

      /* proceed with application processing */
      while (notDone)
      {
          if (EventIsPending > 0)
          {
              /* retrieve a TSAPI event */
              acsGetEventPoll (acsHandle, ...etc...);
```

```
                EventIsPending = 0;
                /* re-enable handler */
                signal (SIGIO, handleEvent);
            }
            /* perform other background processing... */
        }
}
```

TSAPI SDK for Linux

Use this section for information about developing TSAPI applications using Linux. You do not need to be familiar with the CSTA call model or API, but you should read Chapter 3: Control Services.

File locations

For information about installed files see Appendix A of the *Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide*, 02-300543.

Development Platforms

The TSAPI header files in this SDK are compatible with the Linux Compiler. Using another compiler may require you to modify the header files, for example, to account for differences in structure alignment, size of enumerated data types, and so forth.

Linking to the TSAPI Library

The TSAPI for Linux client is implemented as a shared object library, libcsta.so, and follows the standard conventions for library path search and dynamic linking. If libcsta.so is installed in one of the standard directories, it is only necessary to include "-lcsta" in your link step, for example:

```
cc -D_REENTRANT -o myprog myprog.c -lcsta
```

Note:
Applications using private data also need to include `-lattpriv` in the link step.

Using Application Control Services

This section describes how to use application control services (ACS) to retrieve events on Linux. If you are porting code that uses telephony services, you should read this section to get an overview of the differences between Linux and other platforms.

Event Notification:

The acsEventNotify() and acsSetESR() functions are not provided on the Linux platform.

Linux does not directly promote an event-driven programming model, but rather a file-oriented one. To work most effectively in the Linux environment, the TSAPI event stream should appear as a file. The acsGetFile() function returns the file descriptor associated with an ACS stream handle. The returned value may be used like any other file descriptor in an I/O multiplexing call, such as poll() or select(), to determine the availability of TSAPI events.

Important:
Do not perform other I/O or control operations directly on this file descriptor. Doing so may lead to unpredictable results from the TSAPI library.

Receiving Events

This section describes event reception using acsGetEventPoll() and acsGetEventBlock() on Linux.

Blocking Versus Polling:

The acsGetEventBlock() function suspends the calling application until it receives an event. If your application has no other work to perform in the meantime, this is the simplest and most efficient way to receive events from the TSAPI. Typically, however, an application needs to respond to input from the user or other sources, and cannot afford to wait exclusively for TSAPI events. The acsGetEventPoll() function returns control immediately if no event is available, allowing the application to query other input sources or events.

Calling acsGetEventPoll() repetitively can unduly consume processor time and resources, to the detriment of other applications. Instead of polling, consider multiplexing your input sources via the poll() or select() system calls.

Receiving Events From Any Stream:

An application may specify a NULL stream handle when calling acsGetEventPoll() or acsGetEventBlock() to request that the TSAPI Service library return the first event available on any of that application's streams.

Message Trace

To create a log file of TSAPI messages sent to and received from the TSAPI Service, set the shell environment variable CSTATRACE to the pathname of the desired file, prior to starting your application. The log file will be created if necessary, or appended to if it already exists.

Basic TSAPI programming tips

This section provides some basic, TSAPI programming tips on the following topics:

Note:
For more information about designing applications see Advanced TSAPI Programming Techniques on page 44.

Opening and closing streams

This section provides some fundamental TSAPI programming information about opening and closing ACS streams. For information about API Control Services (ACS), see Chapter 3: Control Services on page 61.

- Your application must close all open streams -- preferably by calling acsAbortStream() -- before exiting.
- If you use acsCloseStream(), you must retrieve the ACSCloseStreamConfEvent by calling acsGetEventBlock() or acsGetEventPoll().

  Unless your application needs to process all outstanding events before exiting, use acsAbortStream() instead of acsCloseStream().
- When opening a stream, an application may negotiate with the TSAPI Service to agree upon the version of private data protocol to be used (see Requesting private data on page 169).

Monitoring switch object state changes

Call Control Services allow a client application to control a call or connections on a switch. Although client applications can manipulate switch objects, Call Control Services do not provide Event Reports as objects change state. To monitor switch object state changes (that is, to receive Event Report Services from a switch), a client must request a Monitor Service for an object before it requests Call Control Services for that object.

Client/server roles and the routing service

The CSTA client/server relationship allows for bi-directional services. Both switching and computer applications can assume the role of either client or server.

Currently, Routing Service is the only CSTA service in which the switch application is the client. In all other CSTA services, the computer application is the client. When an application requests a service, a local communications component in the client communicates the request to the server. Each instance of a request creates a new client/server relationship.

An application should open only one stream per advertised service. Each stream carries messages for the application to one advertised service.

The client/server session and the operation invocation model

A client must establish a communication channel to the TSAPI Service before the application can request service from the TSAPI Service. For the TSAPI Service, this communication channel is an API Control Service stream. This stream establishes a session between a TSAPI application (at a client PC) and the server. An application uses the acsOpenStream function to open a stream. The function returns an acsHandle that the application uses to identify the stream. At that time, the application may use the invokeID in some other request.

When a client application requests a CSTA Service, it passes an invokeID that it may use later to associate a response from the server with a specific request. A client's request for service is also called an *operation invocation*. The server replies (via a *service response*) to the client's request with either confirmation (result) or failure (error/rejection) and includes the invokeID in the response.

Some services (such as monitoring a call or device) continue their operation beyond the service response. Since the invokeID no longer identifies the service invocation after an acknowledgment, an additional identifier is necessary for such services. These services return a cross-referenceID in their acknowledgment. The cross-referenceID is a unique value that an application can use to associate event reports with the initiating service request. The cross-reference terminates when the service stops.

Advanced TSAPI Programming Techniques

This section provides you with some programming techniques that are useful for designing desktop oriented applications. It includes the following topics.

Transferring or conferencing a call together with screen pop information

Many desktop applications involve scenarios where an incoming call arrives at a monitored phone, (for example, a claims agent) and the application uses caller information to pop a screen at that desktop. At some point, the claims agent realizes that both the call and the data screen need to be shared with some other person, (for example, a supervisor). The claims agent may need to conference in the supervisor, or may need to transfer the call to the supervisor. In both cases, a similar application running at the supervisor's desktop that is monitoring the supervisor's phone needs to obtain information about the original caller from CSTA events to pop the same screen at the supervisor's desktop.

Before designing a screen pop application, an application designer must first understand the caller information that the TSAPI Service makes available. When an incoming call arrives at a monitored station device, the TSAPI Service provides CSTA Delivered and Established events that contain a variety of caller information:

- **Calling Number** (CSTA parameter) - This parameter contains the calling number, when known. An application may use the calling number to access customer records in a database. The Event Report chapter contains detailed information about the facilities that provide Calling Number.
- **Called Number** (CSTA parameter) - This parameter contains the called number, when known. Often this parameter contains the "DNIS" for an incoming call from the public network. An application may use the called number to pop an appropriate screen when, for example, callers dial different numbers to order different products.
- **Digits Collected by Call Prompting** (Avaya private data) - Integrated systems often route callers to a voice response unit that collects the caller's account number. These voice response units can often be integrated with a Communication Manager Server so that the caller's account number is made available to the monitoring application. An application may use the collected digits to access customer records in a database.
- **User-to-User Information (UUI)** (Avaya private data) - This parameter contains information that some other application has associated with the incoming call. UUI has the important property that it can be passed across certain facilities (PRI) which can be purchased within the public switched network. An application may use the calling number to access customer records in a database.
- **Lookahead Interflow Information** (Avaya private data) - This parameter contains information about the call history of an incoming call that is being forwarded from a remote Communication Manager Server.

CSTA Services Used to Conference or Transfer Calls

The previous section, Transferring or conferencing a call together with screen pop information on page 45), described the caller information that the TSAPI Service makes available. Your next considerations are the various CSTA services that you can use to conference or transfer calls, and the different event contents that result from these services.

The following sections describe two examples of TSAPI service sequences that an application can use to conference or transfer calls.

- Using the Consultation Call Service on page 46
- Emulating Manual Operations on page 47

Using the Consultation Call Service

This example depicts what happens when the Consultation Call Service is used with either the Conference Call service or the Transfer Call Service.

The following steps depict the operations involved.

1. cstaConsultationCall()
2. cstaConferenceCall() or cstaTransferCall()

First, the Consultation Call Service service, cstaConsultationCall(), places an active call on hold and then makes a consultation call (such as the call to the supervisor described in Transferring or conferencing a call together with screen pop information on page 45). Next, the cstaConferenceCall() or cstaTransferCall() conferences or transfers the call.

Unique Advantage of the Consultation Call Service

The unique (and important) attribute of cstaConsultationCall() is that the consultation service associates the call being placed on hold with the consultation call.

An application that monitors the phone receiving the consultation call will see information about the original caller in an Avaya private data item called “Original Call Information” appearing in the CSTA Delivered event.

“Original Call Information” gives an application (such as the supervisor’s) the information necessary to pop a screen using the original caller’s information at the time that the call begins alerting at the consultation desktop.

Note:
Applications that need to pass information about the original caller and have a screen pop when the call alerts at the consultation desktop should use the cstaConsultationCall() service to place those calls.

Emulating Manual Operations

This example depicts what happens when an application emulates a series of manual operations. The following sequence emulates what a user might do manually at a phone to conference or transfer calls.

1. cstaHoldCall();
2. cstaMakeCall();
3. cstaConferenceCall() or cstaTransferCall().

Unlike the Consultation Call service, these operations do not associate any information about the call being placed on hold with the call that is being made. In fact, such an association cannot be made because the calling station may have multiple calls on hold and the TSAPI Service cannot anticipate which of those will actually be transferred.

However, using this sequence of operations does, in some cases, pass information about the original caller in events for the consultation call. This occurs for transferred calls when Transferring or Conferencing a Call Together with Screen Pop Information the transferring party hangs up before the consultation call is answered. This is known as a “blind transfer”.

Notice that when the consultation party answers the blind transfer, there are two parties on the call, the original caller and the consultation party. Therefore, when the calling party answers, TSAPI Service puts information about the original caller in the CSTA Established event. This sequence allows an application monitoring the party receiving the consultation call to pop a screen about the original caller only in the case of a blind transfer and only when the call is answered.

Using Original Call Information to Pop a Screen

When an incoming call arrives at a monitored desktop (the claims agent in the previous example), an application can use any of the caller information described in CSTA Services Used to Conference or Transfer Calls on page 46 to pop a screen. When the application uses cstaConsultationCall() to pass a call to another phone, the TSAPI Service retains the original caller information in a block of private data called “Original Call Information”. (OCI) The TSAPI Service passes OCI in the Delivered and Established events for the consultation call. Thus, an application monitoring the consultation desktop can use any of the original caller information to pop a screen.

Application designers must be aware of the following:

- OCI indicates that the call is not a new call.
- OCI fields are reported with a non-null value only if they are giving historical data from a prior call that is different than the current call. The implications of this on the called and calling fields are as follows:
  - If a called device is the same as the OCI called device, the OCI called device is reported as null.
  - If a calling device is the same as the OCI calling device, the OCI calling device is reported as null.
- Using cstaConsultationCall() is the recommended way of passing calls from desktop to desktop in such a way that the original caller information is available for popping screens.
- The TSAPI Service shares “Original Call Information” with applications using the same AE Server to monitor phones.
- “Original Call Information” can not be shared across different AE Servers.
- When applications use “Original Call Information” to pop screens, the applications monitoring phones for the community of users among which calls are transferred (typically call center or service center agents) must use the same AE Server.
- The TSAPI Service shifts information into the OCI block as the call information changes. For example, since prompted digits do not change because a call is transferred, the original prompted digits may be in the prompted digit private data parameter rather than the “Original Call Information” block.
- Applications using caller information should look first in the “Original Call Information” block. If they find nothing there, they should use the information in the other private data and CSTA parameters.

Note that, for example, if a call passes through monitored VDN A (which collects digits) and then passes through monitored VDN B (which also collects digits) and then is delivered to monitored VDN C, then in the Delivered event we find the digits from VDN A in the Original Call Information for the call and the digits from VDN B in the Collected Digits private data for the call.

Note:

Using this approach, the application will always use the original caller's information to pop the screen regardless of whether it is running at the desktop that first receives the call (the claims agent) or a consultation desktop (the supervisor's desktop).

Using UUI to Pass Information to Remote Applications

In addition to providing “Original Call Information” to allow original caller information to pass among applications using the same AE Server, Communication Manager provides advanced private data features that let an application developer implement an application that passes caller information to applications that do the following:

- monitor stations using different AE Servers
- monitor stations using more than one type of Communication Manager server (for example, a DEFINITY Server or an S8000-series server running Communication Manager)
- reside on a CTI platform at a remote switch that is monitoring stations are connected to it

Since Communication Manager associates User-to-User Information (UUI) with a call within the Communication Manager server, Communication Manager makes the UUI for a call available on all of its CTI links. Additionally, when a Communication Manager server supplies UUI when making a call (such as a consultation call) across PRI facilities in the public switched network, the UUI passes across the public network to the remote Communication Manager server. The remote Communication Manager server then makes this UUI available to applications on its CTI links.

While “Original Call Information” is a way of sharing all caller information across applications using a given AE Server, UUI is the way to share information across a broader CTI application community, including applications running at remote switch sites.

An important decision in the design of an application that works across multiple AE Server, CTI platforms, and remote Communication Manager servers is what information passes between applications in the UUI.

Application designers must be aware of the following:

- Unlike “Original Call Information”, the amount of information that UUI carries is limited.
- Often the UUI is an account number that has been collected by a voice response unit or obtained from a customer database. It might also be the caller’s telephone number. It might be a record or transaction identifier that the application defines.
- In all cases, the application is responsible for copying or entering the information into the call’s UUI. Applications may enter information into a call’s UUI when they make a call, route a call, or drop a call.
- When an application enters information into a call’s UUI, any previous UUI is overwritten.

- Applications that support large and diverse systems must be designed to expect the same kind of information in the UUI, and the same format of information in the UUI. That is, application design must be carefully coordinated when a system includes multiple AE Server, CTI platforms, or Communication Manager servers.

  For example, when an application includes users on one AE Server, as well as users on other AE Servers, CTI platforms, or Communication Manager servers, a designer could use a hybrid approach. Such an approach would combine the best of “Original Call Information” (all of the original caller data) with the advantages of UUI (sharing information across CTI links and remote switches).

Re-registering as a Routing Server after a TCP/IP failure

Beginning with AE Services 4.0, a routing application can reestablish itself after recovering from a near-end TCP/IP outage. When the TSAPI Service receives a subsequent route register request with the same login name, application name and IP address as the original route request, it will discard the old route register request and honor the new request, thereby allowing the application to reinstate itself.

Prior to AE Services 4.0, if a routing application experienced a near-end TCP/IP outage, and it attempted to re-establish itself after recovering, AE Services would deny the request. The application could not be reinstated as the routing server unless you restarted the TSAPI Service.

Based on the network topology, when a network failure occurs, under some circumstances the client may be able to detect the failure and the AE Server will not. In this case, beginning with AES 4.0, the client application is able to re-open a stream and re-register as a routing application, once the network has recovered.

Who can use the new route register request features?

If you have a network configuration with more than one subnet, this recovery feature applies to you. For all other configurations -- with no subnetting or with only one subnet -- the feature does not apply because TSAPI Service will detect the outage, abort the session, and permit another route register request. For a general refresher about routing at the TSAPI level, continue with Routing transactions on page 53.

Routing transactions

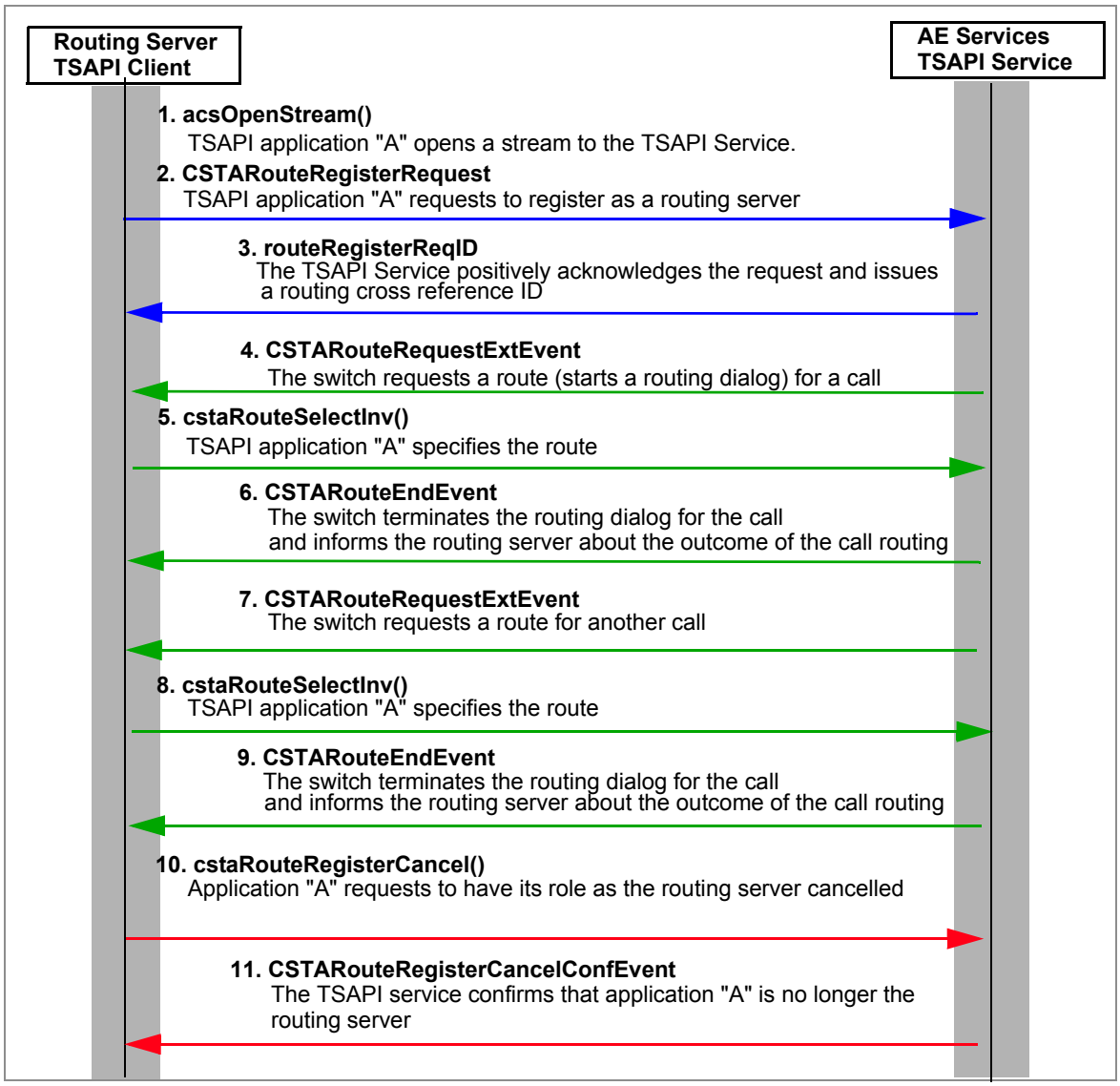
For each routing transaction, the switch sends a **CSTARouteRequestExtEvent** (*route request*) message to the application. The application, in turn, responds to each route request with a **cstaRouteSelectInv()** (*route select*), which specifies a destination. A transaction is completed when the switch responds with a **CSTARouteEndEvent**. The TSAPI Service does not impose a limit on the number of transactions (route requests) from the switching domain. For an illustration, see Figure 1: Routing Cycle on page 54.

1. **acsOpenStream( )** -The routing application opens a stream to the TSAPI Service. The application provides a login ID and application name.
2. **CSTARouteRegisterRequest** - the TSAPI application requests to register as a routing server

   Note:
   The TSAPI Service allows only one application to register as the routing server for a specific switch (routing device). As long as a routing session is active, all other route register requests will be denied with a universal failure event (CSTAUniversalFailureConfEvent). Figure 2: Routing cycle -- demonstrating rule of "one routing application at a time" on page 56 illustrates how this is enforced.
3. **routeRegisterReqID** -The TSAPI Service positively acknowledges the request and issues a routing cross reference ID
4. **CSTARouteRequestExtEvent** -the switch requests a route (starts a routing dialog) for the call.
5. **cstaRouteSelectInv()** -The application specifies the route.
6. **CSTARouteEndEvent -** The switch terminates the routing dialog and informs the routing server about the outcome of the routing.
7. **CSTARouteRequestExtEvent** - The switch requests a route for another call.
8. **cstaRouteSelectInv()** - The application specifies the route.
9. **CSTARouteEndEvent -** The switch terminates the routing dialog and informs the routing server about the outcome of the routing.
10. **acsCloseStream( )** - The application closes the stream, but it may still receive events on the acsHandle for that ACS stream. The application must continue to poll until it receives the ACSCloseStreamConfEvent so that the system releases all stream resources. The stream remains open until the application receives the ACSCloseStreamConfEvent.
11. **ACSCloseStreamConfEvent** - The application receives the ACSCloseStreamConfEvent so that the system releases all stream resources.

In terms of the routing cycle, a service interruption becomes a factor after Step 3, when the TSAPI Service acknowledges the application as the routing server (**routeRegisterReqID)**. For an illustration of the failure and recovery scenario, see Figure 3: Routing scenario demonstrating TCP/IP failure and recovery on page 57 .

Figure 1: Routing Cycle



When an application re-registers as a routing server

When an application (Application A, for example) experiences a near-end TCP/IP outage, the TSAPI Service might not detect the outage. From the viewpoint of the TSAPI service, Application A is the routing server with an open stream, until Application A sends a cstaRoutRegisterCancel() request to close the stream.

Up until AE Services 4.0, the only way Application A could reinstate itself was by restarting the TSAPI Service, thereby closing the stream.

Beginning with AE Services 4.0, if Application A experiences a near-endTCP/IP outage, it can reinstate itself by sending another route register request to the TSAPI Service, once the network has recovered. When the TSAPI Service receives this route register request from Application A, the TSAPI service recognizes that the stream for the new route register request was opened with the same login name, application name and IP address as the original route request, so it discards the original route registration request and re-registers the application as the routing server.

Figure 2: Routing cycle -- demonstrating rule of "one routing application at a time"

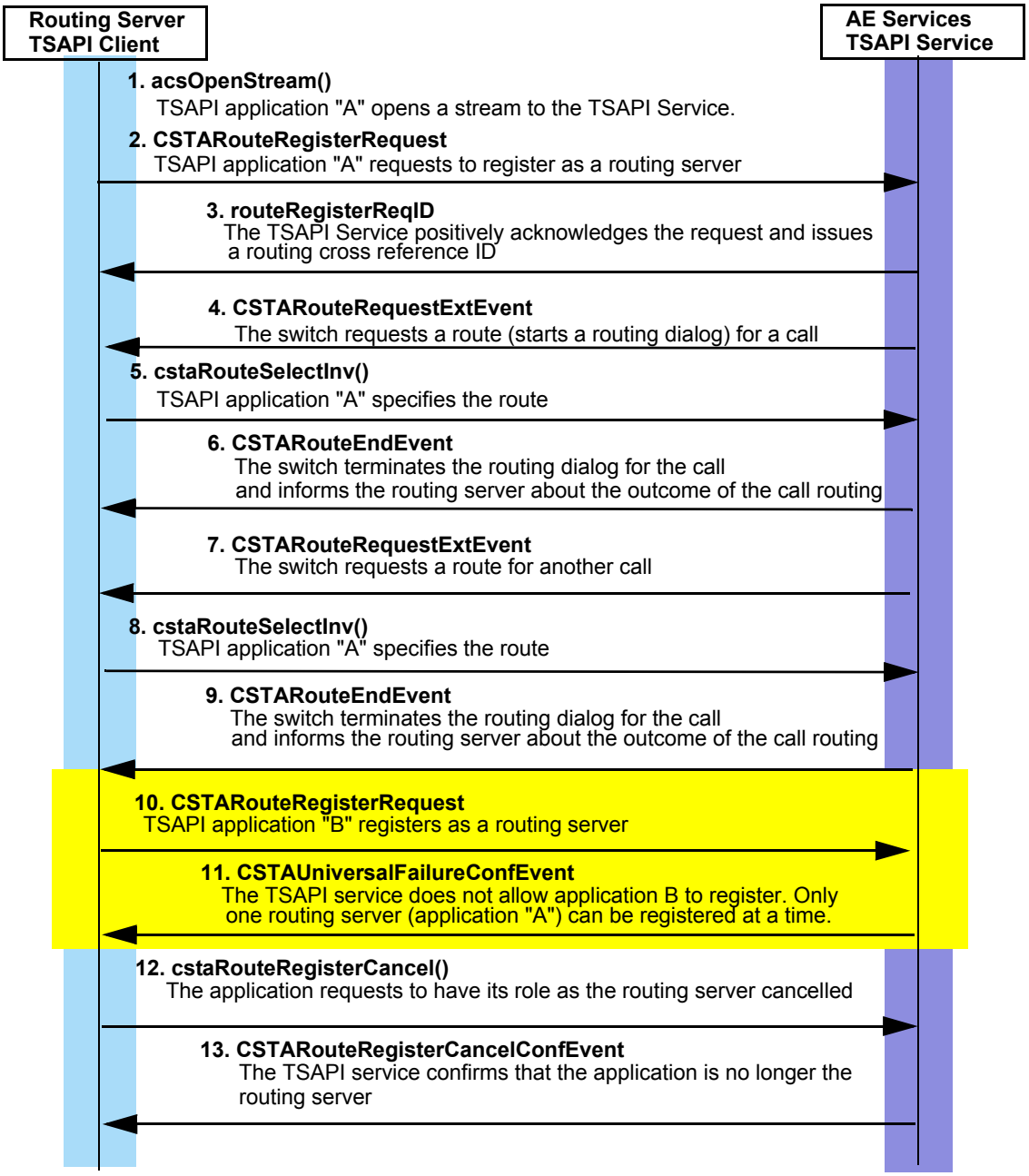
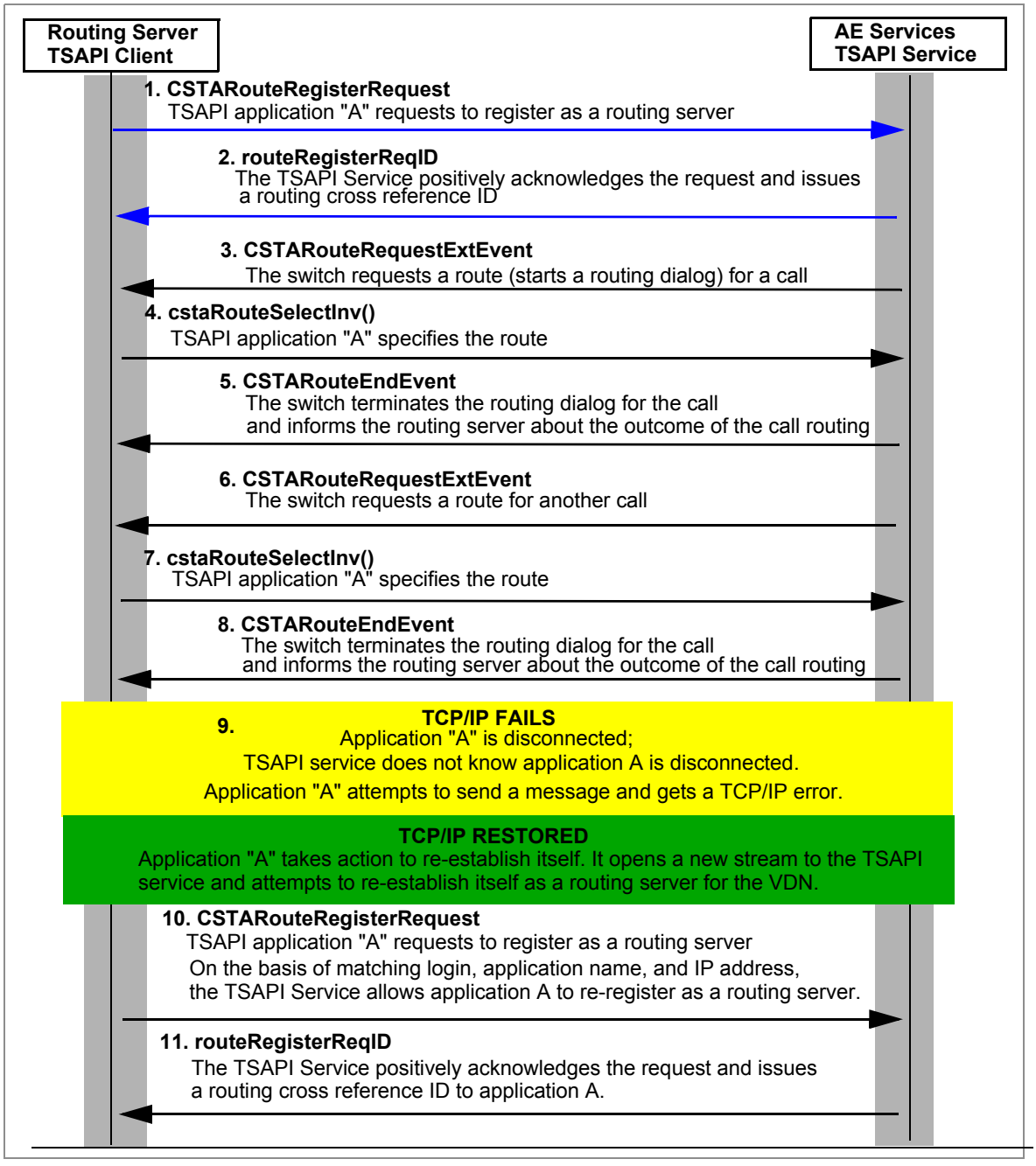


Figure 3: Routing scenario demonstrating TCP/IP failure and recovery



Server-side programming considerations

This section describes the effect that server-side events can have on applications. It includes the following topics.

Multiple AE server considerations

Due to system capacity limitations, care must be taken when using more than one Application Enablement Services server (AE Server) for one Communication Manager server.

Note:
AE Server, in the context of this document, refers to an Application Enablement Services server running the TSAPI Service.

- The simultaneous cstaMonitorDevice() requests on one station device are limited to two per Communication Manager server. A maximum of two AE Servers can monitor the same station at the same time.
- The simultaneous cstaMonitorCallsViaDevice() monitor requests on one ACD device (VDN or ACD split) are limited to one per Communication Manager server. A maximum of one AE Server can monitor the same ACD device at a time.
- A call may pass through an ACD device monitored by one AE Server and be redirected to another ACD device monitored by another AE Server. The former will lose the event reports of that call after Diverted Event Report. Similar cases can result when two calls that are monitored by cstaMonitorCallsViaDevice() requests from different AE Server are merged (transfer or conference operations or requests) into one.

CTI Link Availability

If a link to a Communication Manager server becomes unavailable, all monitors or controls using that link terminate. New monitors or feature requests will be made across any remaining links to the Communication Manager server.

During initialization, the TSAPI Service advertises for each Communication Manager server configured with a link in even if the link to the Communication Manager server is not in service. If an application makes an open stream request and there is no link available to the TSAPI Service, the application will receive an ACS Universal Failure with code (DRIVER_LINK_UNAVAILABLE).

If the link to a Communication Manager server becomes unavailable, any previously opened streams remain open until the application closes them or the TSAPI Service is stopped. The application will not receive a message indicating that the link is unavailable unless the application has used cstaSysStatStart to request system status event reporting.

If a CTI link to a Communication Manager server goes down, the TSAPI Service sends:

- a CSTA Universal Failure event for each outstanding request (cstaMakeCall(), etc.). An outstanding CSTA request is one that has not yet received a confirmation event. The error code is set to RESOURCE_OUT_OF_SERVICE (34). The client should re-issue the request. If the link has become available again, the request will succeed. If the link is still unavailable, the client will continue to receive RESOURCE_OUT_OF_SERVICE (34) and should assume service is unavailable.
- a CSTA Monitor Ended event for any previously established monitor requests. The cause will be EC_NETWORK_NOT_OBTAINABLE (21). The client should re-establish the monitor request. If other links are available, the monitor request will be honored. If no other links are available, the client will receive a CSTA Universal Failure with the error code set to RESOURCE_OUT_OF_SERVICE (34) and should assume service to the switch is unavailable.
- a Route End event for any active Route Select dialogue. The client need do nothing.
- a Route Register Abort event will be sent to the application when the TSAPI link for the registered routing device is down. The application could make use of System and Link Status Notification (see Chapter 13: System Status Service Group on page 751) to determine when the link comes back up. If the application wants to continue the routing service after the CTI link is up, it must issue a cstaRouteRegisterReq() to re-establish a routing registration session for the routing device.

The system status services and events provide private data that inform applications of the status of the multiple links to a Communication Manager server. Refer to (see Chapter 13: System Status Service Group on page 751).

Chapter 3: Control Services

This chapter describes the control services that are provided by the Telephony Services Application Programming Interface (TSAPI).

Note:
In the context of this chapter the term TSAPI refers to TSAPI at the interface level as opposed to TSAPI at the service level (the AE Services TSAPI Service). To make this distinction clear, this chapter uses the term TSAPI interface.

This chapter includes the following topics.

Control Services provided by TSAPI

The TSAPI interface, provides two kinds of control services:

- Application Programing Interface (API) Control Services, or ACS
- CSTA Control Services.

Applications use ACS to manage their interactions with Telephony Services. ACS functions manage the interface, while CSTA functions provide the CSTA services.

API Control Services

Applications use API Control Services (ACS) to do the following:

- Open an ACS stream with the AE Services TSAPI Service (the TSAPI service provides CSTA services)
- Close an ACS stream
- Block or poll for events.
- Initialize an operating system event notification facility. For example, on a Windows client, this initializes an Event Service Routine (ESR)
- Get a list of available advertised services
- Select a private data version for use on the stream
- (Beginning with AE Services 4.1) Query an ACS stream for its service name
- (Beginning with AE Services 4.1) Control the interval at which the TSAPI Service sends heartbeat events to the client.

CSTA Control Services

Applications use the CSTA Control Services to do the following:

- Query for the CSTA Services available on an open ACS stream, see cstaGetAPICaps( ) on page 120
- Query for a list of Devices that CSTA Services can monitor, control or route for on an open ACS stream, see cstaGetDeviceList( ) on page 125
- Query to determine if CSTA Call/Call Monitoring is available on an open ACS stream, see cstaQueryCallMonitor( ) on page 129.

Opening, Closing and Aborting an ACS stream

To access the TSAPI Service, an application must open an ACS stream (or session). This stream establishes a logical link between the application and call processing software on the switch. The application requests CSTA services (such as making a call) over the stream. The TSAPI Service provides ACS streams.

The TSAPI service can be set up to do security checking to ensure that an application receives CSTA services only for permitted devices. Each application must open an ACS stream before it requests any services.

The system advertises CSTA services to applications. An application opens an ACS stream to use an advertised service. Each stream carries messages for the application to one advertised service.

Opening an ACS stream

Here is the sequence for opening an ACS stream.

1. The application calls acsOpenStream( ).

   acsOpenStream( ) is a request to establish an ACS stream with a Telephony Server. The **acsOpenStream( )** function returns an ***acsHandle*** to the application. The application will use this ***acsHandle*** to access the ACS stream (make requests and receive events).

2. The application receives an ACSOpenStreamConfEvent event message that corresponds to the acsOpenStream( ) request.

   The application waits for a corresponding **ACSOpenStreamConfEvent** with the ***acsHandle*** returned by the **acsOpenStream()** request. The application should not request services on the ACS stream until it receives this corresponding **ACSOpenStreamConfEvent**.

After an application successfully receives the **ACSOpenStreamConfEvent**, it may request CSTA Services such as Device (telephone) monitoring.

Important:
The application should always check the **ACSOpenStreamConfEvent** to ensure that the ACS stream has been successfully established before making any CSTA Service requests.

An application is responsible for releasing its ACS stream(s). To release the system resources associated with an ACS stream, the application may either close the stream or abort the stream. Failing to release the resources may corrupt the client system, resulting in client failure.

Important:
An ***acsHandle*** is a local process identifier and should not be shared across processes.

When TSAPI Client configuration file specifies Alternate Tlinks

When the TSAPI Client configuration file specifies Alternate Tlinks, the username and password specified by the application in the acsOpenStream() request should be configured identically for each AE Server. For more information about alternate Tlinks, see the *Avaya MultiVantage Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide*, 02-300543.

Closing an ACS stream

Here is the sequence for closing an ACS stream

1. The application calls acsCloseStream( ) to initiate the orderly shutdown of an ACS stream.

   After the application calls **acsCloseStream**( ) to close an ACS stream, the application may not request any further services on that stream. The **acsCloseStream**( ) function is a non-blocking call. The application passes an *acsHandle* indicating which ACS stream to close. Although the application can not make requests on that stream, the *acsHandle* remains valid until the application receives the corresponding **ACSCloseStreamConfEvent.**

   Important:
   After an application calls **acsCloseStream**( ), it may still receive events on the *acsHandle* for that ACS stream. The application must continue to poll until it receives the **ACSCloseStreamConfEvent** so that the system releases all stream resources. The stream remains open until the application receives the **ACSCloseStreamConfEvent**.

2. The application receives an ACSCloseStreamConfEvent event message that corresponds to the acsCloseStream( ) request.

   An **ACSCloseStreamConfEvent** indicates that the *acsHandle* for the stream is no longer valid and that the system has freed all system resources associated with the ACS stream. The last event the application will receive on the ACS stream is the **ACSCloseStreamConfEvent**. Closing an ACS stream terminates any CSTA call control sessions on that stream. Terminating CSTA call control sessions in this way does not affect the switch processing of controlled calls. The application can no longer control them on this stream.

Aborting an ACS stream

Here is a description of what happens when an ACS stream aborts.

- The application calls **acsAbortStream( )**.

  An application may use **acsAbortStream**( ) to unilaterally (and synchronously) terminate an ACS stream when

  - it does not require confirmation of successful stream closure, and
  - it does not need to receive any events that may be queued for it on that stream.

The application passes an ***acsHandle*** indicating which ACS stream to abort. The **acsAbortStream**( ) function is non-blocking and returns to the application immediately. When **acsAbortStream**( ) returns, the *acsHandle* is invalid (unlike **acsCloseStream**( )). The system frees all resources associated with the aborted ACS stream, including any events queued on this stream. Aborting an ACS stream terminates any CSTA call control on that stream. Aborting CSTA call control in this way does not affect the switch processing of controlled calls. It terminates the application's control of them on this stream. There is no confirmation event for an **acsAbortStream**( ) call.

Sending CSTA Requests and Responses

After an application opens an ACS stream (including reception of the **ACSOpenStreamConfEvent()** it may request CSTA services and receive events. In each service request, the application passes the *acsHandle* of the stream over which it is making the request.

Each service request requires an *invokeID* that the system will return in the confirmation event (or failure event) for the function call. Since applications may have multiple requests for the same service outstanding within the same ACS stream, *invokeIDs* provide a way to match the confirmation event (or failure event) to the corresponding request. When an application opens an ACS stream, it specifies (for that stream) whether it will:

- generate and manage *invokeID*s internally, or
- have the TSAPI library generate unique *invokeID* for each service request.

Once an application specifies this *invokeID* type for an ACS stream, the application cannot change *invokeID* type for the stream.

In general, having the TSAPI library generate unique *invokeID*s simplifies application design. However, when service requests correspond to entries in a data structure, it may simplify application design to use indexes into the data structure as *invokeID*s. Application-generated *invokeIDs* might also point to Windows handles. Application-generated *invokeIDs* may take on any 32-bit value.

Receiving Events

When an application successfully opens an ACS stream, the TSAPI Library queues the **ACSOpenStreamConfEvent** event message for the application. To receive this event, and subsequent event messages, the application must use one of two event reception methods:

- a blocking mode, which blocks the application from executing until an event becomes available. Blocking is appropriate in threaded or preemptive operating system environments only (Windows XP or 2000, for example).
- a non-blocking mode that returns control to the application regardless of whether an event is available.

> **Important:**
> Blocking on event reports may be appropriate for applications that monitor a Device and only require processing cycles when an event occurs. However, there may be operating system specific implications. For example, if a Windows application blocks waiting for CSTA events, then it cannot process events from its Windows event queue.

Regardless of the mode that an application uses to receive events, it may elect to receive an event either from a designated ACS stream (that it opened) or from any ACS stream (that it has opened). TSAPI gives the application the events in chronological order from the selected stream(s). Thus, if the application receives events from all ACS streams, then it receives the events in chronological order from all the Streams.

Blocking Event Reception

Here is the sequence for blocking event reception.

1. The application calls acsGetEventBlock( )

   acsGetEventBlock( ) function gets the next event or blocks if no events are available. The application passes an ***acsHandle*** parameter containing the handle of an open ACS stream or a zero value (indicating that it desires events from any open ACS stream).

2. acsGetEventBlock( ) returns when an event is available.

Non-Blocking Event Reception

Here is the sequence for blocking event reception.

1. The application calls acsGetEventPoll( )

   Applications use **acsGetEventPoll( )** to poll for events at their own pace. An application calls **acsGetEventPoll**( ) any time it wants to process an event. The application passes an *acsHandle* containing the handle of an open ACS stream or a zero value (indicating that it desires events from any open ACS stream). In addition, the *numEvents* parameter tells the application how many events are on the queue.

2. acsGetEventPoll( ) returns immediately

   a. If one or more events are available on the ACS stream, **acsGetEventPoll**( ) returns the next event from the specified stream (or from any stream, if the application selected that option).

   b. When the event queue is empty, the function returns immediately with a “no message” indication.

Important:
The application must receive events (using either the blocking or polling method) frequently enough so that the event queue does not overflow. TSAPI will stop acknowledging messages from the Telephony Server when the queue fills up, ultimately resulting in a loss of the stream. When a message is available, it does not matter which function an application uses to retrieve it.

In some operating system environments (Windows, Windows NT), an application can use an ***Event Service Routine*** **(ESR)** to receive asynchronous notification of arriving events. The ESR mechanism *notifies* the application of arriving events. It does not remove the events from the event queue. The application must use **acsGetEventBlock( )** or **acsGetEventPoll( )** to *receive* the message. The application can use an ESR to trigger a specific action when an event arrives in the event queue (i.e. post a Windows message for the application). See the manual page for **acsSetESR**( ) for more information about ESR use in specific operating system environments.

TSAPI makes one other event handling function available to applications: **acsFlushEventQueue( )**. An application uses **acsFlushEventQueue( )** to flush all events from an ACS stream event queue (or, if the application selects, from all ACS stream event queues).

Specifying TSAPI versions when you open a stream

As TSAPI evolves over time to support more services, TSAPI will include new functions and event reports. To ensure that applications written to earlier versions of the system will continue to operate with newer TSAPI libraries, TSAPI provides version control.

Currently AE Services supports TSAPI Version 2 only.

Note:
A TSAPI version comprises a set of function calls and events. When a new version of TSAPI is introduced, new names are assigned to TSAPI functions, and new events are assigned to new event type values. It is the programmer's responsibility to ensure that the program uses only TSAPI functions from the appropriate version set.

Providing a list of TSAPI versions in the API version parameter

An application provides a list of the TSAPI versions that it is willing to accept in the API version parameter (***apiVer***) parameter of the open stream function, **acsOpenStream( )**. See acsOpenStream ( ) on page 73.

This parameter contains an ASCII string that is formatted with no spaces, as follows:

```
TSn-n:5
```

where:

TS is a fixed constant (use uppercase characters).

n is a number indicating the TSAPI version

- (hyphen) character indicates a range of versions.

: (colon) character indicates a list of versions.

Example

The following example depicts how an application specifies that it can use TSAPI versions 1 through 3 (1,2, and 3) and version 5.

```
TS1-3:5
```

How the TSAPI version is negotiated

As the TSAPI Service processes the open stream request, it checks to see which of the requested versions it supports. If it cannot support a requested version, it removes that version from the list before passing the request on to the next component. The TSAPI Service opens the stream using the highest (latest) TSAPI version remaining and returns that version to the application in the **ACSOpenStreamConfEvent**. Once a stream is opened, the version is fixed for the duration of the stream.

If the TSAPI service cannot find a suitable version, the open stream request fails and the application receives an **ACSUniversalFailureConfEvent** (see ACS Related Errors on page 810.

The TSAPI Service returns the selected TSAPI version in the ***apiVer*** field of the **ACSOpenStreamConfEvent**. The version begins with the letters **ST** (the **S** and the **T** are intentionally reversed) followed by a single TSAPI version number. If the contents of the apiVer field do not begin with the letters ST, then the application should assume TSAPI version 1.

Requesting private data when you open an ACS stream

Although similar in format to the TSAPI version negotiation, the Private Data version negotiation is independent of TSAPI version negotiation.

- When an application opens a stream to the TSAPI service, the application needs to indicate to the TSAPI Service what private data version or versions the application supports. See Requesting private data on page 169.
- If an application does not support private data, the application uses a NULL pointer to indicate to the TSAPI Service that it does not support private data. This lets you save the LAN bandwidth that the private data will consume. See Applications that do not use private data on page 170.

Querying for Available Services

Applications can use the **acsEnumServerNames( )** function to obtain a list of the advertised service names. The presence of an advertised service name in the list does not mean that it is available.

ACS functions and confirmation events

This section describes the following API Control Services (ACS) functions and confirmation events.

acsOpenStream ( )

An application uses **acsOpenStream( )** to open an ACS stream to an advertised service, which for the TSAPI Service, is a TLINK. An application needs an ACS stream to access other ACS Control Services or CSTA Services. Thus, an application must call **acsOpenStream( )** before requesting any other ACS or CSTA service -- **acsOpenStream( )** immediately returns an *acsHandle*; a confirmation event arrives later.

As of Release 4.1.0, AE Services introduces the Alternate Tlinks feature. This feature provides the TSAPI Service with the ability select an alternate Tlink if the Tlink specified in the **acsOpenStream()** request is not available when the procedure is executed. To effect the alternate Tlink selection you must specify the alternate Tlinks in the TSAPI Configuration file.

For information about setting up the TSAPI Configuration file, see the *Avaya MultiVantage. Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide*, 02-300543.

Syntax

```
#include <acs.h>
#include <acslimit.h>
#include <csta.h>

RetCode_t acsOpenStream(
  ACSHandle_t        *acsHandle,          /* RETURN */
  InvokeIDType_t     invokeIDType,        /* INPUT */
  InvokeID_t         invokeID,            /* INPUT */
  StreamType_t       streamType,          /* INPUT */
  ServerID_t         *serverID,           /* INPUT */
  LoginID_t          *loginID,            /* INPUT */
  Passwd_t           *passwd,             /* INPUT */
  AppName_t          *applicationName,    /* INPUT */
  Level_t            acsLevelReq          /* INPUT */
  Version_t          *apiVer,             /* INPUT */
  unsigned short     sendQSize,           /* INPUT */
  unsigned short     sendExtraBufs,       /* INPUT */
  unsigned short     recvQSize,           /* INPUT */
  unsigned short     recvExtraBufs        /* INPUT */
  PrivateData_t      *privateData);       /* INPUT */
```

Parameters

acsHandle

The acsOpenStream( ) service request returns this value that identifies the ACS stream that was opened. TSAPI sets this value so that it is unique to the ACS stream. Once acsOpenStream( ) is successful, the application must use this acsHandle in all other function calls to TSAPI on this stream. If acsOpenStream( ) is successful, TSAPI guarantees that the application has a valid acsHandle. If acsOpenStream( ) is not successful, then the function return code gives the cause of the failure.

invokeIDType

The application sets the type of invoke identifiers used on the stream being opened. The possible types are as follows:

- Application-Generated invokeIDs (**APP_GEN_ID**)

  When **APP_GEN_ID** is selected, the application will provide an invokeID with every TSAPI function call that requires an invokeID. TSAPI will return the supplied invokeID value to the application in the confirmation event for the service request. Application-generated invokeID values can be any 32-bit value.

- Library generated invokeIDs (**LIB_GEN_ID**)

  When **LIB_GEN_ID** is selected, the ACS Library will automatically generate an invokeID and will return its value upon successful completion of the function call. The value will be the return from the function call (RetCode_t). Library-generated invoke IDs are always in the range 1 to 32767.

invokeID

The application supplies this handle for matching the acsOpenStream( ) service request with its confirmation event. An application supplies a value for invokeID only when the ***invokeIDtype*** parameter is set to **APP_GEN_ID**. TSAPI ignores the ***invokeID*** parameter when invokeIDtype parameter is set to **LIB_GEN_ID**.

streamType

The application provides the type of stream in streamType. The possible values are:

- **ST_CSTA** - identifies a request as a CSTA call control stream. This stream can be used for TSAPI service requests and responses which begin with the prefix **csta** or **CSTA**.
- **ST_OAM** - requests an OAM stream. (The AE Services TSAPI Service does not support this value).

serverID The application provides a null-terminated string of maximum size **ACS_MAX_SERVICEID**. This string contains the name of an advertised service (in ASCII format). The application must ensure that the serverID provides services of the type given in the streamType parameter.

Notes:

When multiple AE Servers are used as alternates, the username and password specified by the application in the acsOpenStream() request should be configured identically for each AE Server.

When the TSAPI Configuration file specifies alternate server IDs (Tlinks) for this serverID, the stream may be opened to one of the alternate server IDs instead of the requested server ID. An AES 4.1 application that requires the actual serverID for the stream can call acsGetServerID().

loginID The application provides a pointer to a null terminated string of maximum size **ACS_MAX_LOGINID**. This string contains the login ID of the user requesting access to the advertised service given in the *serviceID* parameter.

passwd The application provides a pointer to a null terminated string of maximum size **ACS_MAX_PASSWORD**. This string contains the password of the user given *loginID*.

applicationName The application provides a pointer to a null terminated string of maximum size **ACS_MAX_APPNAME**. This string contains an application name. The system uses the application name on certain administration and maintenance status displays.

acsLevelReq This version of TSAPI ignores this parameter.

apiVer An application uses this parameter to specify the TSAPI version.

This parameter contains a string beginning with the characters “TS” followed by an ASCII encoding of one or more version numbers. An application may use the “-” (hyphen) character to specify a range of versions and the “:” (colon) character to separate a list of versions. For example, the string “TS1-3:5” specifies that the application is willing to accept TSAPI versions 1, 2, 3, or 5.

NOTE: All applications should specify Version 2 for the TSAPI Service.

See Specifying TSAPI versions when you open a stream on page 69.

sendQSize The application specifies in *sendQsize* the maximum number of outgoing messages the TSAPI client library will queue before returning ACSERR_QUEUE_FULL. If the application supplies a zero (0) value, then a default queue size will be used.

NOTE: The Linux TSAPI client library does not use the *sendQSize* parameter.

sendExtraBufs	The application specifies the number of additional packet buffers TSAPI allocates for the send queue. If *sendExtraBufs* is set to zero (0), the number of buffers is equal to the queue size (i.e., one buffer per message). If you expect messages to exceed the size of a network packet, a reasonable expectation if you use private data extensively, be sure to allocate additional buffers. Also, if your application frequently returns the error ACSERR_NOBUFFERS, it indicates that the application has not allocated enough buffers. **NOTE:** The Linux TSAPI client library does not use the *sendExtraBufs* parameter.
recvQSize	The application specifies the maximum number of incoming messages the TSAPI Client Library queues before it ceases acknowledgment to the Telephony Server. TSAPI uses a default queue size when recvQSize is set to zero (0). **NOTE:** The Linux TSAPI client library does not use the *recvQSize* parameter.
recvExtraBufs	The application specifies the number of additional packet buffers that TSAPI allocates for the receive queue. If recvExtraBufs is set to zero (0), the number of buffers is equal to the queue size (i.e., one buffer per message). If messages will exceed the size of a network packet, as in the case where private data is used extensively, or the application frequently sees **ACSERR_STREAM_FAILED**, then the application does not use recvExtraBufs to allocate enough buffers. **NOTE:** The Linux TSAPI client library does not use the *recvExtraBufs* parameter.
privateData	The application uses this parameter to provide a pointer to a data structure that contains any implementation-specific initialization. For the TSAPI Service this pointer is used to specify Avaya Private Data. The TSAPI protocol does not interpret the data in this structure. The application provides a NULL pointer when Private Data is not present. No private data on an open stream request is a request to the TSAPI Service not to send any private data. For information about negotiating private data versions, see Requesting private data on page 169.

Return Values

acsOpenStream( ) returns the following values depending on whether the application is using library or application-generated invoke identifiers:

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

The application should always check the ACSOpenStreamConfEvent message to ensure that the Telephony Server has acknowledged the acsOpenStream( ) request.

acsOpenStream( ) returns the following negative error conditions:

- ACSERR_APIVERDENIED - The requested API version (apiVer) is invalid or the client library does not support it.
- ACSERR_BADPARAMETER - One or more of the parameters is invalid.
- ACSERR_NODRIVER - No TSAPI Client Library Driver was found or installed on the system.
- ACSERR_NOSERVER - The advertised service (serverID) is not available in the network.
- ACSERR_NORESOURCE - There are insufficient resources to open a ACS stream.
- ACSERR_SSL_INIT_FAILED - This return value indicates that a secure connection could not be opened because there was a problem initializing the OpenSSL library.
- ACSERR_SSL_CONNECT_FAILED - This return value indicates that a stream could not be opened because there was a problem establishing an SSL connection to the server. It may be that the server failed to provide a certificate, or that the server certificate is not signed by a trusted Certificate Authority.
- ACSERR_SSL_FQDN_MISMATCH - This return value indicates that a stream could not be opened because the FQDN in the server certificate does not match the expected FQDN.

  Note that an existing application that is configured to use a secure Tlink will not recognize these values: ACSERR_SSL_INIT_FAILED, ACSERR_SSL_CONNECT_FAILED, and ACSERR_SSL_FQDN_MISMATCH.
- acsOpenStream() may also return ACSERR_STREAM_FAILED if the application attempts to open a stream to a secure (encrypted) Tlink but the TSAPI client library (Release 4.0.x or earlier) does not support secure client connections.

Comments

An application uses **acsOpenStream**( ) to open a network or local communication channel (ACS stream) with an advertised service (TSAPI Service). The stream will establish an ACS client/server session between the application and the server. The application can use the ACS stream to access all the server-provided services (for example **cstaMakeCall**, **cstaTransferCall**, etc.). The **acsOpenStream**( ) function returns an *acsHandle* for the stream. The application uses the *acsHandle* to wait for a **ACSOpenStreamConfEvent**. The application uses the **ACSOpenStreamConfEvent** to determine whether the stream opened successfully. The application then uses the *acsHandle* in any further requests that it sends over the stream. An application should only open one stream for any advertised service.

When an application calls **acsOpenStream**( ) the call may block for up to ten (10) seconds while TSAPI obtains names and addresses from the network Name Server.

Applications should not open multiple streams to the same advertised service since this results in inefficient use of system resources.

Application Notes

The TSAPI Service supports a single CTI link to Avaya Communication Manager. Each advertised service name is unique on the network.

The TSAPI interface guarantees that the **ACSOpenStreamConfEvent** is the first event the application will receive on ACS stream if no errors occurred during the ACS stream initialization process.

The application is responsible for terminating ACS streams. To do so, an application either calls **acsCloseStream**( ) (and receives the **ACSCloseStreamConfEvent),** or calls **acsAbortStream**( ). It is imperative that an application close all active stream(s) during its exit or cleanup routine in order to free resources in the client and server for other applications on the network.

The application must be prepared to receive an **ACSUniversalFailureConfEvent** (for any stream type)**, CSTAUniversalFailureConfEvent** (for a CSTA stream type) or an **ACSUniversalFailureEvent** (for any stream type) anytime after the **acsOpenStream**( ) function completes. These events indicate that a failure has occurred on the stream.

With the Alternate Tlinks feature, the stream may be opened to a different advertised service than the advertised service that was specified in the **acsOpenStream()** request. For more information, see acsGetServerID() on page 103.

ACSOpenStreamConfEvent

This event is generated in response to the **acsOpenStream**( ) function and provides the application with status information about the request to open an ACS stream with the TSAPI Service. The application may only perform the ACS functions **acsEventNotify**( )**, acsSetESR**( ), **acsGetEventBlock**( )**, acsGetEventPoll**( ), and **acsCloseStream**( ) on an *acsHandle* until this confirmation event has been received.

Syntax

The following structure shows only the relevant portions of the unions for this message. For more information, see “CSTA Data Types,” Chapter 10 of the *Application Enablement Services TSAPI Programmer’s Reference*, 02-300545.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
                InvokeID_t    invokeID;
                union
                {
                        ACSOpenStreamConfEvent_t   acsopen;
                } u;
        }  acsConfirmation;
    } event;
} CSTAEvent_t;

typedef struct ACSOpenStreamConfEvent_t
{
    Version_t apiVer;
    Version_t libVer;
    Version_t tsrvVer;
    Version_t drvrVer;
} ACSOpenStreamConfEvent_t;
```

Parameters

acsHandle	This is the handle for the ACS stream.
eventClass	This is a tag with the value ACSCONFIRMATION, which identifies this message as an ACS confirmation event.
eventType	This is a tag with the value ACS_OPEN_STREAM_CONF, which identifies this message as an ACSOpenStreamConfEvent.
invokeID	This parameter specifies the requested instance of the function or event. It is used to match a specific function request with its confirmation events.
apiVer	This parameter indicates which version of the API was granted. The version begins with the letters “ST” (the “S” and the “T” are intentionally reversed. Note that the application supplied string had the letters in the order “TS”) followed by a single TSAPI version number. If the contents of the **apiVer** field do not begin with the letters “ST”, then the application should assume TSAPI version 1. See Specifying TSAPI versions when you open a stream on page 69.
libVer	This parameter indicates which version of the Library is running.
tsrvVer	This parameter indicates which version of the TSAPI Service is running.
drvrVer	This parameter indicates which version of the TSAPI Service Driver is running.

Comments

This message is an indication that the ACS stream requested by the application via the **acsOpenStream**( ) function is available to provide communication with the Telephony Server. The application may now request call control services from the Telephony Server on the acsHandle identifying this ACS stream. This message contains the Telephony Services API, TSAPI Client Library, TSAPI Service, and TSAPI Service Driver versions, and any Private data returned by the Telephony Server.

The Private Data in the **ACSOpenStreamConfEvent** indicates what vendor and version Private Data the PBX driver will provide on the stream. In the Private Data, the ***vendor*** field will contain the vendor name and the ***data*** field in the Private_Data_t structure contains a one byte discriminator PRIVATE_DATA_ENCODING followed by an ASCII string identifying the version of the private data that will be supplied.

Application Notes

The **ACSOpenStreamConfEvent** is guaranteed to be the first event on the ACS stream the application will receive if no errors occurred during the ACS stream initialization.

With the Alternate Tlinks feature, the stream may be opened to a different advertised service than the advertised service that was specified in the **acsOpenStream()** request. For more information, see acsGetServerID() on page 103.

acsCloseStream( )

This function closes an ACS stream to the Telephony Server. The application will be unable to request services from the Telephony Server after the **acsCloseStream**( ) function has returned. The *acsHandle* is valid on this stream after the **acsCloseStream**( ) function returns, but can only be used to receive events via the **acsGetEventBlock**( ) or **acsGetEventPoll**( ) functions. The application must receive the **ACSCloseStreamConfEvent** associated with this function call to indicate that the ACS stream associated with the specified *acsHandle* has been terminated and to allow stream resources to be freed.

Syntax

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

RetCode_t acsCloseStream (
    ACSHandle_t     acsHandle,        /* INPUT */
    InvokeID_t      invokeID,         /* INPUT */
    PrivateData_t   *privateData);    /* INPUT */
```

Parameters

acsHandle - This is the handle for the active ACS stream which is to be closed. Once the confirmation event associated with this function returns, the handle is no longer valid.

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

privateData - This points to a data structure which defines any implementation-specific information needed by the server. The data in this structure is not interpreted by the API Client Library and can be used as an escape mechanism to provide implementation specific commands between the application and the Telephony Server.

Return Values

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

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

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

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

acsCloseStream( ) returns the negative error conditions below:

ACSERR_BADHDL - This indicates that the *acsHandle* being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

Comments

Once this function returns, the application must also check the **ACSCloseStreamConfEvent** message to ensure that the ACS stream was closed properly and to see if any Private Data was returned by the server.

No other service request will be accepted to the specified *acsHandle* after this function successfully returns. The handle is an active and valid handle until the application has received the **ACSCloseStreamConfEvent**.

Application Notes

The Client is responsible for receiving the **ACSCloseStreamConfEvent** which indicates resources have been freed.

The application must be prepared to receive multiple events on the ACS stream after the **acsCloseStream**( ) function has completed, but the **ACSCloseStreamConfEvent** is guaranteed to be the last event on the ACS stream.

Only the **acsGetEventBlock**( ) and **acsGetEventPoll**( ) functions can be called after the **acsCloseStream**( ) function has returned successfully.

ACSCloseStreamConfEvent

This event is generated in response to the **acsCloseStream**( ) function and provides information regarding the closing of the ACS stream The *acsHandle* is no longer valid after this event has been received by the application, so the **ACSCloseStreamConfEvent** is the last event the application will receive for this ACS stream.

Syntax

The following structure shows only the relevant portions of the unions for this message. See the **TSAPI Specification** for a complete description of the event structure.

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


typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
            struct
            {
            InvokeID_t invokeID;
                    union
                    {
                            ACSCloseStreamConfEvent_t   acsclose;
                    } u;
            } acsConfirmation;
    } event;
} CSTAEvent_t;

typedef struct ACSCloseStreamConfEvent_t
    {
    Nulltype   null;
} ACSCloseStreamConfEvent_t;
```

Parameters

acsHandle - This is the handle for the opened ACS stream.

eventClass - This is a tag with the value **ACSCONFIRMATION**, which identifies this message as an ACS confirmation event.

eventType - This is a tag with the value **ACS_CLOSE_STREAM_CONF**, which identifies this message as an **ACSCloseStreamConfEvent**.

invokeID - This parameter specifies the requested instance of the function. It is used to match a specific **acsCloseStream**( ) function request with its confirmation event.

Comments

This message indicates that the ACS stream to the Telephony Server has closed and that the associated *acsHandle* is no longer valid. This message contains any Private data returned by the Telephony Server.

ACSUniversalFailureConfEvent

This event can occur at any time in place of a confirmation event for any of the CSTA functions which have their own confirmation event and indicates a problem in the processes of the requested function. The ACSUniversalFailureConfEvent does not indicate a failure or loss of the ACS stream with the TSAPI Service. If the ACS stream has failed, then an ACSUniversalFailureEvent (unsolicited version of this confirmation event) is sent to the application, see ACSUniversalFailureEvent on page 110.

Syntax

The following structure shows only the relevant portions of the unions for this message. See ***ACS Data Types*** on page 114 and CSTA Event Data Types on page 132 for a complete description of the event structure.

```
typedef struct
{
    ACSHandle_t acsHandle;
    EventClass_t eventClass;
    EventType_t eventType;
} ACSEventHeader_t;
typedef struct
{
    ACSEventHeader_t   eventHeader;
    union
    {
        struct
        {
                union
                {
                ACSUniversalFailureConfEvent_t     failureEvent;
                } u;
        } acsConfirmation;
    } event;
} CSTAEvent_t;


typedef struct
{
       ACSUniversalFailure_t error;
} ACSUniversalFailureConfEvent_t;
```

Parameters

acsHandle - This is the handle for the ACS stream.

eventClass - This is a tag with the value **ACSCONFIRMATION**, which identifies this message as an ACS confirmation event.

eventType - This is a tag with the value **ACS_UNIVERSAL_FAILURE_CONF**, which identifies this message as an **ACSUniversalFailureConfEvent**.

error - This parameter indicates the cause value for the failure of the original Telephony request. These cause values are the same set as those shown for **ACSUniversalFailureEvent**.

Comments

This event will occur anytime when a non-telephony problem (no memory, TSAPI Service Security check failed, etc.) in processing a Telephony request is encountered and is sent in place of the confirmation event that would normally be received for that function (i.e., **CSTAMakeCallConfEvent** in response to a **cstaMakeCall**( ) request). If the problem which prevents the telephony function from being processed is telephony based, then a **CSTAUniversalFailureConfEvent** will be received instead.

acsAbortStream( )

This function unilaterally closes an ACS stream to the Telephony Server. The application will be unable to request services from the Telephony Server or receive events after the **acsAbortStream**( ) function has returned. The *acsHandle* is invalid on this stream after the **acsAbortStream**( ) function returns. There is no associated confirmation event for this function.

Syntax

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

RetCode_t acsAbortStream (
  ACSHandle_t      acsHandle,     /* INPUT */
  PrivateData_t    *privateData); /* INPUT */
```

Parameters

acsHandle - This is the handle for the active ACS stream which is to be closed. There is no confirmation event for this function. Once this function returns success, the ACS stream is no longer valid.

privateData - This points to a data structure which defines any implementation-specific information needed by the server. The data in this structure is not interpreted by the API Client Library and can be used as an escape mechanism to provide implementation specific commands between the application and the Telephony Server.

Return Values

This function always returns zero (0) if successful.

The following are possible negative error conditions for this function:

ACSERR_BADHDL - This indicates that the ***acsHandle*** being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

Comments

Once this function returns, the ACS stream is dismantled and the *acsHandle* is invalid.

acsGetEventBlock( )

This function is used when an application wants to receive an event in a **Blocking** mode. In the **Blocking** mode, the application will be blocked until there is an event from the ACS stream indicated by the *acsHandle*. If the *acsHandle* is set to zero (0), then the application will block until there is an event from ***any*** ACS stream opened by this application. The function will return after the event has been copied into the applications data space.

Syntax

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

RetCode_t    acsGetEventBlock (
             ACSHandle_t       acsHandle,          /* INPUT */
             void              *eventBuf,          /* INPUT */
             unsigned short    *eventBufSize,      /* INPUT/RETURN */
             PrivateData_t     *privateData,       /* RETURN */
             unsigned short    *numEvents);        /* RETURN */
```

Parameters

acsHandle - This is the value of the unique handle to the opened ACS stream. If a handle of zero (0) is given, then the next message on any of the open ACS streams for this application is returned.

eventBuf - This is a pointer to an area in the application address space large enough to hold one incoming event that is received by the application. This buffer should be large enough to hold the largest event the application expected to receive. Typically the application will reserve a space large enough to hold a CSTAEvent_t.

eventBufSize - This parameter indicates the size of the user buffer pointed to by *eventBuf*. If the event is larger the *eventBuf,* then this parameter will be returned with the size of the buffer required to receive the event. The application should call this function again with a larger buffer.

privateData - This parameter points to a buffer which will receive any private data that accompanies this event. The *length* field of the PrivateData_t structure must be set to the size of the *data* buffer. If the application does not wish to receive private data, then *privateData* should be set to NULL.

numEvents - The library will return the number of events queued for the application on this ACS stream (not including the current event) via the *numEvents* parameter. If this parameter is NULL, then no value will be returned.

Return Values

This function returns a positive acknowledgment or a negative error condition (< 0). There is no confirmation event for this function. The positive return value is:

ACSPOSITIVE_ACK

The function completed successfully as requested by the application, and an event has been copied to the application data space. No errors were detected.

Possible local error returns are (negative returns):

ACSERR_BADHDL

This indicates that the acsHandle being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

ACSERR_UBUFSMALL

The user buffer size indicated in the *eventBufSize* parameter was smaller than the size of the next available event for the application on the ACS stream. The *eventBufSize* variable has been reset by the API Library to the size of the next message on the ACS stream. The application should call **acsGetEventBlock**( ) again with a larger buffer. The ACS event is still on the API Library queue.

Comments

The **acsGetEventBlock**( ) and **acsGetEventPoll**( ) functions can be intermixed by the application. For example, if bursty event message traffic is expected, an application may decide to block initially for the first event and wait until it arrives. When the first event arrives the blocking function returns, at which time the application can process this event quickly and poll for the other events which may have been placed in queue while the first event was being processed. The polling can be continued until a **ACSERR_NOMESSAGE** is returned by the polling function. At this time the application can then call the blocking function again and start the whole cycle over again.

There is no confirmation event for this function.

Application Notes

The application is responsible for calling the **acsGetEventBlock**( ) or **acsGetEventPoll**( ) function frequently enough that the API Client Library does not overflow its receive queue and refuse incoming events from the Telephony Server.

The TSAPI Service may send the application internal events that are not exposed to the application. When one of these events arrives, a Linux application that uses poll() or select() with the file descriptor of an ACS stream will be notified that input is available. However, because the event has been consumed by the TSAPI library, a subsequent call to acsGetEventBlock() will block. For this reason, such applications should only call acsGetEventPoll().

acsGetEventPoll( )

This function is used when an application wants to receive an event in a **Non-Blocking** mode. In the **Non-Blocking** mode the oldest outstanding event from any active ACS stream will be copied into the applications data space and control will be returned to the application. If no events are currently queued for the application, the function will return control immediately to the application with an error code indicating that no events were available.

Syntax

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

RetCode_t    acsGetEventPoll (
             ACSHandle_t        acsHandle,         /* INPUT */
             void               *eventBuf,         /* INPUT */
             unsigned short     *eventBufSize,     /* INPUT/RETURN */
             PrivateData_t      *privateData,      /* RETURN */
             unsigned short     *numEvents;        /* RETURN */
```

Parameters

acsHandle - This is the value of the unique handle to the opened ACS stream. If a handle of zero (0) is given, then the next message on any of the open ACS streams for this application is returned.

eventBuf - This is a pointer to an area in the application address space large enough to hold one incoming event that is received by the application. This buffer should be large enough to hold the largest event the application expected to receive. Typically the application will reserve a space large enough to hold a CSTAEvent_t.

eventBufSize - This parameter indicates the size of the user buffer pointed to by *eventBuf*. If the event is larger the *eventBuf,* then this parameter will be returned with the size of the buffer required to receive the event. The application should call this function again with a larger buffer.

privateData - This parameter points to a buffer which will receive any private data that accompanies this event. The *length* field of the PrivateData_t structure must be set to the size of the *data* buffer. If the application does not wish to receive private data, then *privateData* should be set to NULL.

numEvents - The library will return the number of events queued for the application on this ACS stream (not including the current event) via the *numEvents* parameter. If this parameter is NULL, then no value will be returned.

Return Values

This function returns a positive acknowledgment or a negative error condition (< 0). There is no confirmation event for this function. The positive return value is:

ACSPOSITIVE_ACK

The function completed successfully as requested by the application, and an event has been copied to the application data space. No errors were detected.

Possible local error returns are (negative returns):

ACSERR_BADHDL

This indicates that the acsHandle being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

ACSERR_NOMESSAGE

There were no messages available to return to the application.

ACSERR_UBUFSMALL

The user buffer size indicated in the *eventBufSize* parameter was smaller than the size of the next available event for the application on the ACS stream. The *eventBufSize* variable has been reset by the API Library to the size of the next message on the ACS stream. The application should call **acsGetEventPoll**( ) again with a larger buffer. The ACS event is still on the API Library queue.

Comments

When this function is called, it returns immediately, and the user must examine the return code to determine if a message was copied into the user's data space. If an event was available, the function will return **ACSPOSITIVE_ACK**.

If no events existed on the ACS stream for the application, this function will return **ACSERR_NOMESSAGE**.

The **acsGetEventBlock**( ) and **acsGetEventPoll**( ) functions can be intermixed by the application. For example, if bursty event message traffic is expected, an application may decide to block initially for the first event and wait until it arrives. When the first event arrives the blocking function returns, at which time the application can process this event quickly and poll for the other events which may have been placed in queue while the first event was being processed. The polling may continue until the **ACSERR_NOMESSAGE** is returned by the polling function. At this time the application can then call the blocking function again and start the whole cycle over again.

There is no confirmation event for this function.

Application Notes

The application is responsible for calling the **acsGetEventBlock**( ) or **acsGetEventPoll**( ) function frequently enough that the API Client Library does not overflow its receive queue and refuses incoming events from the Telephony Server.

The TSAPI Service may send the application internal events that are not exposed to the application. When one of these events arrives, a Linux application that uses poll() or select() with the file descriptor of an ACS stream will be notified that input is available. However, because the event has been consumed by the TSAPI library, a subsequent call to acsGetEventPoll() will return ACSERR_NOMESSAGE. The application should not treat this as an error condition.

acsGetFile() (Linux)

The **acsGetFile**( ) function returns the Unix file descriptor associated with an ACS stream. This is to enable multiplexing of input sources via, for example, the **`poll()`** system call.

Syntax

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

RetCode_t acsGetFile (ACSHandle_t acsHandle);
```

Parameters

acsHandle - This is the value of the unique handle to the opened ACS stream whose Unix file descriptor is to be returned.

Return Values

This function returns either a Unix file descriptor greater than or equal to zero (0), or ***ACSERR_BADHDL*** if the ***acsHandle*** being used is not a valid handle for an active ACS stream.

Application Notes

The **acsGetFile**() function returns the UNIX file descriptor used by an ACS stream. This enables an application to simultaneously block on the stream and any other file-oriented input sources by using **poll()**, **select()**, **XtAddInput()** or similar multiplexing functions. The application should never perform any direct I/O operations on this file descriptor.

The TSAPI Service may send the application internal events that are not exposed to the application. When one of these messages arrives on the stream, a call to poll() or select() will return, indicating that input is available on the stream's file descriptor. A subsequent call to acsGetEventBlock() will block, however, because the event has been consumed by the TSAPI client library. For this reason, such applications should only call **acsGetEventPoll().**

There is no confirmation event for this function.

acsSetESR( ) (Windows)

The **acsSetESR**( ) function also allows the application to designate an Event Service Routine (***ESR***) that will be called when an incoming event is available.

Syntax

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

typedef void (*EsrFunc)(unsigned long esrParam)

RetCode_t acsSetESR (
    ACSHandle_t      acsHandle,
    EsrFunc          esr,
    unsigned long    esrParam,
    Boolean          notifyAll);
```

Parameters

acsHandle - This is the value of the unique handle to the opened stream for which this ESR routine will apply. Only one ESR is allowed per active acsHandle.

esr - This is a pointer to the ESR (the address of a function). An application passes a NULL pointer to clear an existing ESR*.*

esrParam - This is a user-defined parameter which will be passed to the ESR when it is called*.*

notifyAll - If this parameter is **TRUE** then the ESR will be called for every event*.* If it is **FALSE** then the ESR will only be called each time the receive queue becomes non-empty, i.e. the queue count changes from zero (0) to one (1). This option may be used to reduce the overhead of notification.

Return Values

This function returns a positive acknowledgment or a negative error condition (< 0). There is no confirmation event for this function. The positive return value is:

ACSPOSITIVE_ACK

The function completed successfully as requested by the application. No errors were detected.

Possible local error returns are (negative returns):

ACSERR_BADHDL

This indicates that the acsHandle being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

Comments

The ESR mechanism can be used by the application to receive an asynchronous notification of the arrival of an incoming event from the ACS stream. The ESR routine will receive one user-defined parameter. The ESR should **not** call TSAPI functions, or the results will be indeterminate. The ESR should note the arrival of the incoming event, and complete its operation as quickly as possible. The application must still call **acsGetEventBlock( )** or **acsGetEventPoll( )** to retrieve the event from the Client API Library queue.

Use **acsSetESR**( ) with care. The ESR code will be executed in the context of a background thread created by the API Client Library, **not** an application thread.

If there are already events in the receive queue waiting to be retrieved when **acsSetESR( )** is called, the ***esr*** will be called for each of them.

The ***esr*** in the **acsSetESR( )** function will replace the current ESR maintained by the API Client Library. A NULL ***esr*** will disable the current ESR mechanism.

There is no confirmation event for this function.

acsEventNotify( ) (Windows)

The **acsEventNotify**( ) function allows a Win32 application to request that a message be posted to its application queue when an incoming ACS event is available.

Syntax

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

RetCode_t acsEventNotify (
      ACSHandle_t  acsHandle,
      HWND         hwnd,
      UINT         msg,
      Boolean      notifyAll);
```

Parameters

acsHandle - This is the value of the unique handle to the opened ACS stream for which event notification messages will be posted.

hwnd - This is the handle of the window which is to receive event notification messages. If this parameter is NULL, event notification is disabled*.*

msg - This is the user-defined message to be posted when an incoming event becomes available*.* The *wParam* and *lParam* parameters of the message will contain the following members of the ACSEventHeader_t structure:

```
wParam          acsHandle
HIWORD(lParam)  eventClass
LOWORD(lParam)  eventType
```

notifyAll - If this parameter is **TRUE** then a message will be posted for every event*.* If it is **FALSE** then a message will only be posted each time the receive queue becomes non-empty, i.e. the queue count changes from zero (0) to one (1). This option may be used to reduce the overhead of notification, or the likelihood of overflowing the application's message queue.

Return Values

This function returns a positive acknowledgment or a negative error condition (< 0). There is no confirmation event for this function. The positive return value is:

ACSPOSITIVE_ACK

The function completed successfully as requested by the application. No errors were detected.

Possible local error returns are (negative returns):

ACSERR_BADHDL

This indicates that the acsHandle being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

Application Notes

This function only enables *notification* of an incoming event. Use **acsGetEventPoll**( ) to actually retrieve the complete event structure.

If there are already events in the receive queue waiting to be retrieved when **acsEventNotify**( ) is called, a message will be posted for each of them.

The rate of notifications may be reduced by setting ***notifyAll*** to **FALSE**.

There is no confirmation event for this function.

Example

This example uses the **acsEventNotify** function to enable event notification.

```
#define WM_ACSEVENT WM_USER + 99
    // or use RegisterWindowMessage()

long FAR PASCAL
WndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    // declare local variables...

    switch (msg)
    {
    case WM_CREATE:

        // post WM_ACSEVENT to this window
        // whenever an ACS event arrives

        acsEventNotify (acsHandle, hwnd, WM_ACSEVENT, TRUE);

        // other initialization, etc...
        return 0;
    case WM_ACSEVENT:

        // wParam contains an ACSHandle_t
        // HIWORD(lParam) contains an EventClass_t
        // LOWORD(lParam) contains an EventType_t

        // dispatch the event to user-defined
        // handler function here

        return 0;

    // process other window messages...

}
return DefWindowProc (hwnd, msg, wParam, lParam);
```

acsFlushEventQueue( )

This function removes all events for the application on a ACS stream associated with the given handle and maintained by the API Client Library. Once this function returns the application may receive any new events that arrive on this ACS stream.

Syntax

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

RetCode_t ACSFlushEventQueue (ACSHandle_t acsHandle);
```

Parameters

acsHandle - This is the handle to an active ACS stream. If the *acsHandle* is zero (0), then TSAPI will flush all active ACS streams for this application.

Return Values

This function returns a positive acknowledgment or a negative error condition (< 0). There is no confirmation event for this function. The positive return value is:

ACSPOSITIVE_ACK

The function completed successfully as requested by the application. No errors were detected.

Possible local error returns are (negative returns):

ACSERR_BADHDL

This indicates that the acsHandle being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

Comments

Once this function returns the API Client Library will not have any events queued for the application on the specified ACS stream. The application is ready to start receiving new events from the Telephony Server.

There is no confirmation event for this function.

Application Notes

The application should exercise caution when calling this function, since all events from the switch on the associated ACS stream have been discarded. The application has no way to determine what kinds of events have been destroyed, and may have lost events that relay important status information from the switch.

This function does not delete the **ACSCloseStreamConfEvent**, since this function can not be called after the **acsCloseStream**( ) function.

The **acsFlushEventQueue**( ) function will delete all other events queued to the application on the ACS stream. The **ACSUniversalFailureEvent** and the **CSTAUniversalFailureConfEvent**, in particular, will be deleted if they are currently queued to the application.

Do not invoke **acsFlushEventQueue()** while there any outstanding acsSetHeartbeatInterval() requests on the ACS stream. This may cause the client library to close the stream.

acsEnumServerNames( )

This function is used to enumerate the names of all the advertised services of a specified stream type. This function is a synchronous call and has no associated confirmation event.

Syntax

```
#include <acs.h>

typedef Boolean (*EnumServerNamesCB)(
    char      *serverName,
    unsigned long  lParam);

RetCode_tacsEnumServerNames(
    StreamType_t       streamType,
    EnumServerNamesCB  callback ,
    unsigned long      lParam);
```

Parameters

streamType - indicates the type of stream requested. The currently defined stream types are **ST_CSTA** and **ST_OAM**.

callback - This is a pointer to a callback function which will be invoked for *each* of the enumerated server names, along with the user-defined parameter ***lParam***. If the callback function returns **FALSE** (0), enumeration will terminate.

lParam - A user-defined parameter which is passed on each invocation of the callback function.

Return Values

This function returns a positive acknowledgment or a negative error condition (< 0). There is no confirmation event for this function. The positive return value is:

ACSPOSITIVE_ACK

The function completed successfully as requested by the application. No errors were detected.

The following are possible negative error conditions for this function:

ACSERR_UNKNOWN

The request has failed due to unknown network problems.

ACSERR_NOSERVER

The request has failed because the client is using TCP/IP and IP addresses are not configured properly.

Comments

This function enumerates all the known advertised services, invoking the callback function for each advertised service name. The ***serverName*** parameter points to automatic storage; the callback function must make a copy if it needs to preserve this data. Under Windows, the callback function must be exported and its address obtained from **MakeProcInstance**( ).

An active ACS stream is ***NOT*** required to call this function.

acsGetServerID()

Use **acsGetServerID()** to get the server ID (TSAPI link name) of the stream.

When a TSAPI client configuration includes Alternate Tlink entries, an **acsOpenStream()** request may open a stream to a different server ID than the requested server ID. For more information, see the *Avaya MultiVantage Application Enablement Services TSAPI, JTAPI, and CVLAN Client and SDK Installation Guide*, 02-300543.

Use acsGetServerID() to determine the actual server ID for an open stream.

Syntax

```
#include <acs.h>

RetCode_t acsGetServerID(
    ACSHandle_t        acsHandle,                /* INPUT */
    ServerID_t         *serverID),               /* INPUT */
```

Parameters

***acsHandle* :** This is the handle for the active ACS Stream which is being queried.

Return Values

This service returns one of the following values:

ACSPOSITIVE_ACK

The service request was successful.

ACSERR_BADHDL

The ACS handle is not a valid handle for an active ACS Stream.

ACSERR_BADPARAMETER

The ***serverID*** parameter is invalid.

If the service is successful, the client library copies the Tlink name for the stream to the memory pointed to by the serverID parameter.

Application Notes

This function is only available for the Windows and Linux client libraries, version 4.1 and later.

There is no confirmation event for this function.

acsQueryAuthInfo( )

Use **acsQueryAuthInfo**( ) to determine the login and password requirements when opening an ACS stream to a particular advertised CSTA service. This function call places the result of a query in a user-provided structure before returning; there is no confirmation event.

Syntax

```
#include <acs.h>
RetCode_t acsQueryAuthInfo(
     ServerID_t     *serverID,     /* INPUT  */
     ACSAuthInfo_t  *authInfo);    /* RETURN */
```

Parameters

serverID **-** The application provides a null-terminated string of maximum size **ACS_MAX_SERVICEID**. This string contains the name of an advertised CSTA service (in ASCII format).

authInfo **-** The application provides a pointer to a pre-allocated structure into which the **acsQueryAuthInfo**( ) returns authentication information about the CSTA service named in ***serverID***. The ACSAuthInfo_t structure is defined as follows:

```
typedef enum
{
    REQUIRES_EXTERNAL_AUTH = -1,
    AUTH_LOGIN_ID_ONLY = 0,
    AUTH_LOGIN_ID_IS_DEFAULT = 1,
    NEED_LOGIN_ID_AND_PASSWD = 2,
    ANY_LOGIN_ID = 3

} ACSAuthType_t;

typedef struct
{
    ACSAuthType_t  authType;
    LoginID_t      authLoginID;

} ACSAuthInfo_t;
```

Return Values

acsQueryAuthInfo( ) returns the negative error conditions below: **-**

ACSERR_BADPARAMETER

One or more of the parameters is invalid.

ACSERR_NODRIVER

No TSAPI Client Library Driver was found or installed on the system.

ACSERR_NOSERVER

The advertised service (***serverID***) is not available in the network.

ACSERR_NORESOURCE

There are insufficient resources to query the advertised service.

Background

The Telephony Services architecture allows network administrators to grant telephony privileges to users. Depending on the implementation of a telephony server and its client libraries, a user may convince telephony servers of his or her identity – authenticate – by different means.

Version 1 of TSAPI required applications to supply a login name and password when calling **acsOpenStream**( ) – the point at which a telephony server must be convinced of a user's identity.

Version 2 and future versions offer support for multiple types of authentication. A telephony service may still require – or simply accept – a login and password, or it may rely on an external authentication service to establish a user's identity.

The Telephony Services architecture offers support for both methods in any combination.

Usage

Call **acsQueryAuthInfo**( ) to determine the authentication requirements for an advertised service (PBX Driver). The caller must provide the name of the advertised service and a pointer to storage into which **acsQueryAuthInfo**( ) will place the query results.

When an application calls **acsQueryAuthInfo**( ), the application may block while the telephony services library queries the specified service.

Examine ***authInfo.authType*** upon return from **acsQueryAuthInfo**( ) to determine what ***loginID*** and ***passwd*** parameters to supply to **acsOpenStream**( ) for the service queried.

REQUIRES_EXTERNAL_AUTH:

The service specified in the query requires the user to authenticate with an external authentication service before opening a stream. If ***authInfo.authType*** contains this value, **acsOpenStream**( ) will fail for the service queried.

AUTH_LOGIN_ID_ONLY:

The application can only open a stream using the ***loginID*** returned in ***authInfo.authLoginID***.

acsOpenStream( ) will ignore ***passwd*** for the queried service. The ***loginID*** must contain the same value as ***authInfo.authLoginID***. An application should not collect a password from its user for this service.

AUTH_LOGIN_ID_IS_DEFAULT:

The ***loginID*** returned in ***authInfo.authLoginID*** is the default user for this service. If the application subsequently specifies this ***loginID*** or a NULL pointer as ***loginID*** to **acsOpenStream**( ), ***passwd*** will be ignored and may be NULL.

Alternatively, to open a stream as a different user than ***authInfo.authLoginID***, the application must supply ***loginID*** and ***passwd*** to **acsOpenStream( )**.

Note:
An application should take care to not collect a password if its user wants to be identified as ***authInfo.authLoginID***. If an application does not remember the last ***loginID*** selected by its user in a preferences file or other persistent storage, the application should use ***authInfo.authLoginID*** as the default ***loginID*** when prompting its user for login information.

NEED_LOGIN_ID_AND_PASSWD:

The application must supply ***loginID*** and ***passwd*** to **acsOpenStream**( ).

ANY_LOGIN_ID:

The application may supply any ***loginID*** to **acsOpenStream**( ); ***passwd*** should not be collected and will be ignored. Applications should default to ***authInfo.authLoginID*** if it is non-empty.

acsSetHeartbeatInterval()

Use **acsSetHeartbeatInterval()** to set the heartbeat interval. As of AE Services 4.1.0 a TSAPI client can indicate a new heartbeat interval value, per stream, by sending a new acsSetHeartbeatInterval() request to the AE Server with the desired interval value in seconds. (Valid values, are 5 - 60.)

If an invalid heartbeat interval is requested (less than 5 seconds or greater than 60 seconds), then the request shall be rejected. Otherwise, when the TSAPI Service receives the request, it shall change the interval value for the stream and respond with an **ACSSetHeartbeatIntervalConf** event.

Syntax

```
#include <acs.h>

RetCode_t acsSetHeartbeatInterval(
    ACSHandle_t        acsHandle,                /* INPUT */
    InvokeID_t         invokeID,                 /* INPUT */
    unsigned short     heartbeatInterval,        /* INPUT */
    PrivateData_t      *privateData);            /* INPUT */
```

Parameters

acsHandle: This is the handle to an open ACS Stream whose heartbeat interval is to be changed.

invokeID: A value provided by the application to be used for matching a specific instance of a service request with its associated confirmation event. This parameter is only used when the Invoke ID mechanism is set for application-generated invoke IDs in the acsOpenStream() request. The parameter is ignored by the ACS library when the stream is set for library-generated invoke IDs.

privateData: This points to a data structure which defines any implementation-specific information needed by the server. The data in this structure is not interpreted by the client library and can be used as an escape mechanism to provide implementation specific commands between the application and the Telephony Server. For AES 4.1, the value of this parameter is ignored.

Return Values

If the stream has library-generated invoke IDs and the function call completes successfully, acsSetHeartbeatInterval() returns a positive value, i.e. the invoke ID. If the function call fails, a negative (<0) value is returned.

If the stream has application-generated invoke IDs and the function call completes successfully, acsSetHeartbeatInterval() returns ACSPOSITIVE_ACK. If the function call fails, a negative (<0) value is returned.

acsSetHeartbeatInterval() has the following negative return values:

ACSERR_BADHDL - The ACS handle is not a valid handle for an active ACS Stream.

Application Notes

This function is only available for the Windows and Linux client libraries, version 4.1 and later.

An application should not invoke acsFlushEventQueue() while there are outstanding acsSetHeartbeatInterval() requests.

The TSAPI Service will only send a heartbeat event to the TSAPI Client if no other events have been sent on a stream within the last heartbeat interval. Thus, the TSAPI heartbeat mechanism will not unduly create unnecessary traffic on the local area network.

Beginning with AE Services 4.1.0, the TSAPI Service sends internal messages, called heartbeat events, to the TSAPI Client at a regular interval, known as the heartbeat interval. If the TSAPI Client library determines that it has not received any events for two heartbeat intervals, then it assumes a network failure has occurred, closes the ACS stream, and notifies the application with an ACSUnsolicited ACSUniversalFailureEvent.

The default heartbeat interval is 20 seconds."

ACSSetHeartbeatIntervalConfEvent

This event is generated in response to the acsSetHeartbeatInterval() function and provides the current heartbeat interval for the ACS Stream.

Syntax

```
typedef struct
{
    ACSHandle_t acsHandle;
    EventClass_t eventClass;
    EventType_t eventType;
} ACSEventHeader_t;
typedef struct
{
    ACSEventHeader_teventHeader;
    union
    {
        struct
        {
    InvokeID_t invokeID;
    union
    {
    ACSSetHeartbeatIntervalConfEvent_t
    acssetheartbeatinterval;
    } u;
        } acsConfirmation;
    } event;
} CSTAEvent_t;


typedef struct ACSSetHeartbeatIntervalConfEvent_t
{
       unsigned short heartbeatInterval;
} ACSSetHeartbeatIntervalConfEvent_t;
```

Parameters

acsHandle: This is the handle of the ACS Stream whose heartbeat interval has been changed

eventClass: This is a tag with the value ***ACSCONFIRMATION***, which identifies this message as an ACS confirmation event.

eventType: This is a tag with the value ***ACS_SET_HEARTBEAT_INTERVAL_CONF***, which identifies this message as an ***ACSSetHeartbeatIntervalConfEvent***.

invokeID: This parameter specifies the requested instance of the function. It is used to match a specific **acsSetHeartbeatInterval()** function request with its confirmation event.

heartbeatInterval: This parameter provides the current heartbeat interval for the ACS Stream.

ACS Unsolicited Events

This section describes unsolicited ACS Status Events.

ACSUniversalFailureEvent

This event can occur at any time (unsolicited) and can indicate, among other things, a failure or loss of the ACS stream with the TSAPI Service.

By contrast, a similarly named event, ACSUniversalFailureConfEvent does not indicate a loss of the ACS stream. See ACS Data Types on page 114 and CSTA Event Data Types on page 132 for a complete description of the event structure.

Syntax

The following structure shows only the relevant portions of the unions for this message. See ***ACS Data Types*** on page 114 and CSTA Event Data Types on page 132 for a complete description of the event structure

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
              union
              {
                ACSUniversalFailureEvent_t failureEvent;
              } u;
        } acsUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    ACSUniversalFailure_t error;
}
ACSUniversalFailureEvent_t;
```

Parameters

acsHandle - This is the handle for the ACS stream.

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

eventType - This is a tag with the value **ACS_UNIVERSAL_FAILURE**, which identifies this message as an **ACSUniversalFailureEvent**.

error - This parameter contains a TSAPI Service operation error (or "cause value"), TSAPI Service security database error, or driver error for the ACS stream given in *acsHandle*.

Note:
Not all of the errors listed below will occur in an ACS Universal Failure message. Some of the errors occur only in error conditions generated by the TSAPI Service.

The possible values are:

```
typedef enum ACSUniversalFailure_t {
       TSERVER_STREAM_FAILED = 0,
       TSERVER_NO_THREAD = 1,
       TSERVER_BAD_DRIVER_ID = 2,
       TSERVER_DEAD_DRIVER = 3,
       TSERVER_MESSAGE_HIGH_WATER_MARK = 4,
       TSERVER_FREE_BUFFER_FAILED = 5,
       TSERVER_SEND_TO_DRIVER = 6,
       TSERVER_RECEIVE_FROM_DRIVER = 7,
       TSERVER_REGISTRATION_FAILED = 8,
       TSERVER_TRACE = 10,
       TSERVER_NO_MEMORY = 11,
       TSERVER_ENCODE_FAILED = 12,
       TSERVER_DECODE_FAILED = 13,
       TSERVER_BAD_CONNECTION = 14,
       TSERVER_BAD_PDU = 15,
       TSERVER_NO_VERSION = 16,
       TSERVER_ECB_MAX_EXCEEDED = 17,
       TSERVER_NO_ECBS = 18,
       TSERVER_NO_SDB = 19,
       TSERVER_NO_SDB_CHECK_NEEDED = 20,
       TSERVER_SDB_CHECK_NEEDED = 21,
       TSERVER_BAD_SDB_LEVEL = 22,
       TSERVER_BAD_SERVERID = 23,
       TSERVER_BAD_STREAM_TYPE = 24,
       TSERVER_BAD_PASSWORD_OR_LOGIN = 25,
       TSERVER_NO_USER_RECORD = 26,
       TSERVER_NO_DEVICE_RECORD = 27,
       TSERVER_DEVICE_NOT_ON_LIST = 28,
       TSERVER_USERS_RESTRICTED_HOME = 30,
```

```
TSERVER_NO_AWAYPERMISSION = 31,
TSERVER_NO_HOMEPERMISSION = 32,
TSERVER_NO_AWAY_WORKTOP = 33,
TSERVER_BAD_DEVICE_RECORD = 34,
TSERVER_DEVICE_NOT_SUPPORTED = 35,
TSERVER_INSUFFICIENT_PERMISSION = 36,
TSERVER_NO_RESOURCE_TAG = 37,
TSERVER_INVALID_MESSAGE = 38,
TSERVER_EXCEPTION_LIST = 39,
TSERVER_NOT_ON_OAM_LIST = 40,
TSERVER_PBX_ID_NOT_IN_SDB = 41,
TSERVER_USER_LICENSES_EXCEEDED = 42,
TSERVER_OAM_DROP_CONNECTION = 43,
TSERVER_NO_VERSION_RECORD = 44,
TSERVER_OLD_VERSION_RECORD = 45,
TSERVER_BAD_PACKET = 46,
TSERVER_OPEN_FAILED = 47,
TSERVER_OAM_IN_USE = 48,
TSERVER_DEVICE_NOT_ON_HOME_LIST = 49,
TSERVER_DEVICE_NOT_ON_CALL_CONTROL_LIST = 50,
TSERVER_DEVICE_NOT_ON_AWAY_LIST = 51,
TSERVER_DEVICE_NOT_ON_ROUTE_LIST = 52,
TSERVER_DEVICE_NOT_ON_MONITOR_DEVICE_LIST = 53,
TSERVER_DEVICE_NOT_ON_MONITOR_CALL_DEVICE_LIST = 54,
TSERVER_NO_CALL_CALL_MONITOR_PERMISSION = 55,
TSERVER_HOME_DEVICE_LIST_EMPTY = 56,
TSERVER_CALL_CONTROL_LIST_EMPTY = 57,
TSERVER_AWAY_LIST_EMPTY = 58,
TSERVER_ROUTE_LIST_EMPTY = 59,
TSERVER_MONITOR_DEVICE_LIST_EMPTY = 60,
TSERVER_MONITOR_CALL_DEVICE_LIST_EMPTY = 61,
TSERVER_USER_AT_HOME_WORKTOP = 62,
TSERVER_DEVICE_LIST_EMPTY = 63,
TSERVER_BAD_GET_DEVICE_LEVEL = 64,
TSERVER_DRIVER_UNREGISTERED = 65,
TSERVER_NO_ACS_STREAM = 66,
TSERVER_DROP_OAM = 67,
TSERVER_ECB_TIMEOUT = 68,
TSERVER_BAD_ECB = 69,
TSERVER_ADVERTISE_FAILED = 70,
TSERVER_TDI_QUEUE_FAULT = 72,
TSERVER_DRIVER_CONGESTION = 73,
TSERVER_NO_TDI_BUFFERS = 74,
TSERVER_OLD_INVOKEID = 75,
TSERVER_HWMARK_TO_LARGE = 76,
TSERVER_SET_ECB_TO_LOW = 77,
TSERVER_NO_RECORD_IN_FILE = 78,
TSERVER_ECB_OVERDUE = 79,
TSERVER_BAD_PW_ENCRYPTION = 80,
TSERVER_BAD_TSERV_PROTOCOL = 81,
```

```
        TSERVER_BAD_DRIVER_PROTOCOL = 82,
        TSERVER_BAD_TRANSPORT_TYPE = 83,
        TSERVER_PDU_VERSION_MISMATCH = 84,
        TSERVER_VERSION_MISMATCH = 85,
        TSERVER_LICENSE_MISMATCH = 86,
        TSERVER_BAD_ATTRIBUTE_LIST = 87,
        TSERVER_BAD_TLIST_TYPE = 88,
        TSERVER_BAD_PROTOCOL_FORMAT = 89,
        TSERVER_OLD_TSLIB = 90,
        TSERVER_BAD_LICENSE_FILE = 91,
        TSERVER_NO_PATCHES = 92,
        TSERVER_SYSTEM_ERROR = 93,
        TSERVER_OAM_LIST_EMPTY = 94,-
        TSERVER_TCP_FAILED = 95,
        TSERVER_TCP_DISABLED = 97,
        TSERVER_REQUIRED_MODULES_NOT_LOADED = 98,
        TSERVER_TRANSPORT_IN_USE_BY_OAM = 99,
        TSERVER_NO_NDS_OAM_PERMISSION = 100,
        TSERVER_OPEN_SDB_LOG_FAILED = 101,
        TSERVER_INVALID_LOG_SIZE = 102,
        TSERVER_WRITE_SDB_LOG_FAILED = 103,
        TSERVER_NT_FAILURE = 104,
        TSERVER_LOAD_LIB_FAILED = 105,
        TSERVER_INVALID_DRIVER = 106,
        TSERVER_REGISTRY_ERROR = 107,
        TSERVER_DUPLICATE_ENTRY = 108,
        TSERVER_DRIVER_LOADED = 109,
        TSERVER_DRIVER_NOT_LOADED = 110,
        TSERVER_NO_LOGON_PERMISSION = 111,
        TSERVER_ACCOUNT_DISABLED = 112,
        TSERVER_NO_NET_LOGON = 113,
        TSERVER_ACCT_RESTRICTED = 114,
        TSERVER_INVALID_LOGON_TIME = 115,
        TSERVER_INVALID_WORKSTATION = 116,
        TSERVER_ACCT_LOCKED_OUT = 117,
        TSERVER_PASSWORD_EXPIRED = 118,
        TSERVER_INVALID_HEARTBEAT_INTERVAL = 119,
        DRIVER_DUPLICATE_ACSHANDLE = 1000,
        DRIVER_INVALID_ACS_REQUEST = 1001,
        DRIVER_ACS_HANDLE_REJECTION = 1002,
        DRIVER_INVALID_CLASS_REJECTION = 1003,
        DRIVER_GENERIC_REJECTION = 1004,
        DRIVER_RESOURCE_LIMITATION = 1005,
        DRIVER_ACSHANDLE_TERMINATION = 1006,
        DRIVER_LINK_UNAVAILABLE = 1007
        DRIVER_OAM_IN_USE = 1008

} ACSUniversalFailure_t;
```

ACS Data Types

This section defines all the data types which are used with the ACS functions and messages and may repeat data types already shown in the ACS Control Functions. Refer to the specific commands for any operational differences in these data types. The ACS data types are type defined in the **acs.h** header file.

Note:
The definition for ACSHandle_t is client platform specific.

This section includes the following topics:

ACS Common Data Types

This section specifies the common ACS data types.

```
typedef int RetCode_t;

        #define ACSPOSITIVE_ACK 0 /* Successful function return */

        /* Error Codes */

        #define ACSERR_APIVERDENIED   -1      /* The API Version
                                               * requested is invalid
                                               * and not supported by
                                               *the API Client Library
                                               */

        #define ACSERR_BADPARAMETER   -2      /* One or more of the
                                              .* parameters is invalid
                                              */

        #define ACSERR_DUPSTREAM      -3      /* This return indicates
                                              * that an ACS stream is
                                              * already established
                                              * with the requested
                                              * Server.
                                              */

        #define ACSERR_NODRIVER       -4      /* This error return
                                               * value indicates that
                                               * no API Client Library * Driver was * found or
                                              *installed on the system
                                              */

        #define ACSERR_NOSERVER       -5      /* the requested Server
                                               * is not present in the network.
                                               */

        #define ACSERR_NORESOURCE     -6      /* there are insufficient
                                               * resourcesto open a
                                               * ACS stream.
                                               */


        #define ACSERR_UBUFSMALL      -7      /* The user buffer size
                                               * was smaller than the
                                               * size of  the next
                                               * available event.
                                               */

        #define ACSERR_NOMESSAGE      -8      /* There were no messages
                                               *available to return to
                                               * the application.
                                               */

         #define ACSERR_UNKNOWN       -9      /* The ACS stream has
```

```
                                * encounteredan
                                * unspecified error.
                                */

#define ACSERR_BADHDL         -10     /* The ACS Handle is
                                       * invalid
                                       */

#define ACSERR_STREAM_FAILED  -11     /* The ACS stream has
                                       * failed due to
                                * network problems.
                                * No further
                                * operations are
                                * possible on this stream.
                                */

#define ACSERR_NOBUFFERS-12/* There were not
                                * enough buffers
                                * available to place
                                * an outgoing message
                                * on the send queue.
                                * No message has been sent.
                                */

#define ACSERR_QUEUE_FULL-13/* The send queue is
                                * full. No message
                                *has been sent.
                                */
#define ACSERR_SSL_INIT_FAILED-14/* This return value indicates that a
                                *stream could not be opened because
                                *initialization of the openSSL library failed.
                                */
#define ACSERR_SSL_CONNECT_FAILED-15/* This return value indicates that a
                                *stream could not be opened because
                                *the SSL connection failed.
                                */
#define ACSERR_SSL_FQDN_MISMATCH-16/* This return value indicates that a
                                *stream could not be opened because
                                *during the SSL handshake, the fully
                                *qualified domain name (FQDN) in the
                                *server certificate did not match the
                                *expected FQDN*/
                                */

typedef unsigned long InvokeID_t;

typedef enum {
   APP_GEN_ID,// application will provide invokeIDs;
       // any 4-byte value is legal
   LIB_GEN_ID // library will generate invokeIDs in
       // the range 1-32767
} InvokeIDType_t;

typedef unsigned short EventClass_t;
```

```
        // defines for ACS event classes

        #defineACSREQUEST 0
        #defineACSUNSOLICITED 1
        #defineACSCONFIRMATION 2

        typedef unsigned short EventType_t;   // event types are
                                              // defined in acs.h
                                              // and csta.h

        typedef char Boolean;
        typedef char Nulltype;

    #define          ACS_OPEN_STREAM    1
    #define          ACS_OPEN_STREAM_CONF    2
    #define          ACS_CLOSE_STREAM    3
    #define          ACS_CLOSE_STREAM_CONF    4
    #define          ACS_ABORT_STREAM    5
    #define          ACS_UNIVERSAL_FAILURE_CONF   6
    #define          ACS_UNIVERSAL_FAILURE    7

    typedef enum StreamType_t {
             ST_CSTA = 1,
             ST_OAM = 2,
    } StreamType_t;

    typedef char ServerID_t[49];

    typedef char LoginID_t[49];

    typedef char Passwd_t[49];

    typedef char AppName_t[21];

typedef enum Level_t {
             ACS_LEVEL1 = 1,
             ACS_LEVEL2 = 2,
             ACS_LEVEL3 = 3,
             ACS_LEVEL4 = 4
    } Level_t;

    typedef char Version_t[21];
```

ACS Event Data Types

This section specifies the ACS data types used in the construction of generic *ACSEvent_t* structures. See specific event types for detailed descriptions of their event structures (see also, CSTA Event Data Types on page 132.

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

typedef struct
{
       ACSEventHeader_t;  eventHeader;
     union
     {
                          .
                          .
                          .
                  ACSUniversalFailureEvent_t failureEvent;
                          .
                          .
                          .
      } u;
     } ACSUnsolicitedEvent;

 typedef struct
     {
     InvokeID_t                    invokeID;
     union
     {
     ACSOpenStreamConfEvent_t              acsopen;
     ACSCloseStreamConfEvent_t             acsclose;
     ACSSetHeartbeatIntervalConfEvent_t    acssetheartbeatinterval;
     ACSUniversalFailureConfEvent_t        failureEvent;
     } u;
     } ACSConfirmationEvent;
```

CSTA control services and confirmation events

This section describes the CSTA functions that the TSAPI Service uses for obtaining information from Communication Manager. For example, the administered switch version, software version, offer Type, server type, as well as the set of devices an application can control, monitor and query. The CSTA control services and confirmation events are as follows:

cstaGetAPICaps( )

Use the AE Services cstaGetAPICaps( ) function to obtain the CSTA API function and event capabilities that are supported on an open CSTA stream. For AE Services the stream could be a local TSAPI Service or a remote TSAPI Service on a network. If a stream provides a CSTA service then it also provides the corresponding CSTA confirmation event.

Syntax

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

RetCode_t    cstaGetAPICaps(
ACSHandle_t  acsHandle,
InvokeID_t   invokeID);
```

Parameters

acsHandle - This is the handle to an active ACS stream. This service will return in its confirmation information about the CSTA services available on this stream.

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

Return Values

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

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

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

The following are possible negative error conditions for this function:

ACSERR_BADHDL

This indicates that the acsHandle being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

Comments

If this function returns with a POSITIVE_ACK, the request has been forwarded to the Telephony Server, and the application will receive an indication of the extent of CSTA service support in the **CSTAGetAPICapsConfEvent**. An active ACS stream is required to the server before this function is called.

The application may use this command to determine which functions and events are supported on an open CSTA stream. This will avoid unnecessary negative acknowledgments from the Telephony Server when a specific API function or event is not supported.

CSTAGetAPICapsConfEvent

This event is in response to the cstaGetAPICaps( ) function and it indicates which CSTA services are available on the CSTA stream.

Syntax

The following structure shows only the relevant portions of the unions for this message. See CSTA Event Data Types on page 132 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
        {
        struct
        {
            InvokeID_t invokeID;
            union
            {
                CSTAGetAPICapsConfEvent_t getAPIcaps;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;


typedef struct CSTAGetAPICapsConfEvent_t {
    short           alternateCall;
    short           answerCall;
    short           callCompletion;
    short           clearCall;
    short           clearConnection;
    short           conferenceCall;
    short           consultationCall;
    short           deflectCall;
    short           pickupCall;
    short           groupPickupCall;
    short           holdCall;
    short           makeCall;
    short           makePredictiveCall;
    short           queryMwi;
    short           queryDnd;
    short           queryFwd;
    short           queryAgentState;
    short           queryLastNumber;
    short           queryDeviceInfo;
    short           reconnectCall;
```

```
short           retrieveCall;
short           setMwi;
short           setDnd;
short           setFwd;
short           setAgentState;
short           transferCall;
short           eventReport;
short           callClearedEvent;
short           conferencedEvent;
short           connectionClearedEvent;
short           deliveredEvent;
short           divertedEvent;
short           establishedEvent;
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;
short           privateEvent;
short           routeRequestEvent;
short           reRoute;
short           routeSelect;
short           routeUsedEvent;
short           routeEndEvent;
short           monitorDevice;
short           monitorCall;
short           monitorCallsViaDevice;
short           changeMonitorFilter;
short           monitorStop;
short           monitorEnded;
short           snapshotDeviceReq;
short           snapshotCallReq;
short           escapeService;
short           privateStatusEvent;
short           escapeServiceEvent;
short           escapeServiceConf;
short           sendPrivateEvent;
short           sysStatReq;
short           sysStatStart;
short           sysStatStop;
```

```
    short           changeSysStatFilter;
    short           sysStatReqEvent;
    short           sysStatReqConf;
    short           sysStatEvent;
} CSTAGetAPICapsConfEvent_t;
```

Parameters

acsHandle - This is the handle for the ACS stream.

eventClass - This is a tag with the value **CSTACONFIRMATION**, which identifies this message as a CSTA confirmation event.

eventType - This is a tag with the value **CSTA_GETAPI_CAPS_CONF**, which identifies this message as an **CSTAGetAPICapsConfEvent.** For information about the private data associated with the CSTAGetAPICapsConfEvent see CSTA Get API Capabilities confirmation structures for Private Data Version 8 on page 171.

getAPIcaps - This structure contains an integer for each possible CSTA capability which indicates whether the capability is supported. A value of 0 indicates the capability is not supported, a positive value indicates that it is supported. Note that different capabilities are supported on different stream versions. This parameter shows what capabilities are supported on the stream where the confirmation has been received. Streams using other versions may support a different capability set.

Comments

This event will provide the application with compatibility information for a specific instance of the TSAPI Service on a command or event basis.

cstaGetDeviceList( )

This is used to obtain the list of Devices that can be controlled, monitored, queried or routed for the ACS stream indicated by the acsHandle.

Syntax

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

RetCode_t      cstaGetDeviceList(
    ACSHandle_t        acsHandle,
    InvokeID_t         invokeID,
    long               index,
    CSTALevel_t        level)
```

Parameters

acsHandle - This is the handle to an active ACS stream.

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

index - The security data base could contain a large number of devices that a user has privilege over, so this API call will return only **CSTA_MAX_GETDEVICE** devices in any one **CSTAGetDeviceListConfEvent,** which means several calls to cstaGetDeviceList() may be necessary to retrieve all the devices**.** ***Index*** should be set of -1 the first time this API is called and then set to the value of ***Index*** returned in the confirmation event. ***Index*** will be set back to -1 in the **CSTAGetDeviceListConfEvent** which contains the last batch of devices.

level - This parameter specifies the class of service for which the user wants to know the set of devices that can be controlled via this ACS stream. ***level*** must be set to one of the following:

```
typedef enum CSTALevel_t {
        CSTA_HOME_WORK_TOP = 1,
        CSTA_AWAY_WORK_TOP = 2,
        CSTA_DEVICE_DEVICE_MONITOR = 3,
        CSTA_CALL_DEVICE_MONITOR = 4,
        CSTA_CALL_CONTROL = 5,
        CSTA_ROUTING = 6,
        CSTA_CALL_CALL_MONITOR = 7
} CSTALevel_t;
```

Note:
The ***level*** CSTA_CALL_CALL_MONITOR is not supported by the **CSTAGetDeviceList()** call. To determine if an ACS stream has permission to do call/call monitoring, use the API call **CSTAQueryCallMonitor( ).**

Return Values

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

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

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

The following are possible negative error conditions for this function:

ACSERR_BADHDL

This indicates that the acsHandle being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

CSTAGetDeviceListConfEvent

This event is in response to the cstaGetDeviceList( ) function and it provide a list of the devices which can be controlled for the indicated ACS Level. It is also possible to receive an **ACSUniversalFailureConf** event in response to a cstaGetDeviceList() call.

Syntax

The following structure shows only the relevant portions of the unions for this message. See ***ACS Data Types*** on page 114 and CSTA Event Data Types on page 132 for a complete description of the event structure.

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

typedef struct
{
       ACSEventHeader_t eventHeader;
       union
       {
       struct
           {
               InvokeID_t invokeID;
               union
               {
                  CSTAGetDeviceListConfEvent_t getDeviceList;
               } u;
           } cstaConfirmation;
       } event;
} CSTAEvent_t;

 typedef enum SDBLevel_t {

    NO_SDB_CHECKING = -1,
    ACS_ONLY = 1,
    ACS_AND_CSTA_CHECKING = 0
 } SDBLevel_t;

 typedef struct CSTAGetDeviceList_t {
    long            index;
    CSTALevel_t     level;
 } CSTAGetDeviceList_t;

 typedef struct DeviceList {
    short           count;
    DeviceID_t      device[20];
 } DeviceList;

 typedef struct CSTAGetDeviceListConfEvent_t {
        SDBLevel_t      driverSdbLevel;
```

```
        CSTALevel_t     level;
        long            index;
        DeviceList      devList;
} CSTAGetDeviceListConfEvent_t;
```

Parameters: ***acsHandle -*** This is the handle for the ACS stream.

eventClass - This is a tag with the value **CSTACONFIRMATION**, which identifies this message as an ACS confirmation event.

eventType - This is a tag with the value **CSTA_GET_DEVICE_LIST_CONF**, which identifies this message as an **CSTAGetDeviceListConfEvent**.

invokeID - This parameter specifies the requested instance of the function. It is used to match a specific function request with its confirmation events.

driverSdbLevel - This parameter indicates the Security Level with which the Driver registered. Possible values are:

- NO_SDB_CHECKING - Not Used.
- ACS_ONLY - Check ACSOpenStream requests only
- ACS_AND_CSTA_CHECKING - Check ACSOpenStream and all applicable CSTA messages

If the SDB database is disabled by administration, and the driver registered with SDB level ACS_AND_CSTA_CHECKING, the TSAPI Service will return the adjusted (effective) SDB checking level of ACS_ONLY. No CSTA checking can be done because there is no database of devices to use for checking the CSTA messages.

index

This parameter indicates to the client application the current index the TSAPI Service is using for returning the list of devices. The client application should return this value in the next call to CSTAGetDeviceList to continue receiving devices. A value of (-1) indicates there are no more devices in the list.

devlist

This parameter is a structure which contains an array of ***DeviceID_t*** which contain the devices for this stream.

cstaQueryCallMonitor( )

This is used to determine the if a given ACS stream has permission to do call/call monitoring in the security database.

Syntax

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

RetCode_t  cstaQueryCallMonitor(
    ACSHandle_t  acsHandle,
    InvokeID_t   invokeID)
```

Parameters

acsHandle - This is the handle to an active ACS stream.

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

Return Values

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

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

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

The following are possible negative error conditions for this function:

ACSERR_BADHDL - This indicates that the acsHandle being used is not a valid handle for an active ACS stream. No changes occur in any existing streams if a bad handle is passed with this function.

CSTAQueryCallMonitorConfEvent

This event is in response to the cstaQueryCallMonitor( ) function and it provide a list of the devices which can be controlled for the indicated ACS Level.

Syntax

The following structure shows only the relevant portions of the unions for this message. See ***ACS Data Types*** on page 114 and CSTA Event Data Types on page 132 for a complete description of the event structure.

```
    typedef struct
                  {
                  ACSHandle_t acsHandle;
                  EventClass_t eventClass;
                  EventType_t eventType;
    } ACSEventHeader_t;
typedef struct
    {
    ACSEventHeader_teventHeader;
    union
    {
                  struct
                  {
                  InvokeID_t invokeID;
                  union
                  {
                    CSTAQueryCallMonitorConfEvent_t  queryCallMonitor;
} u;
} cstaConfirmation;
    } event;
    } CSTAEvent_t;

typedef struct CSTAQueryCallMonitorConfEvent_t {
    Boolean         callMonitor;
} CSTAQueryCallMonitorConfEvent_t;
```

Parameters

acsHandle - This is the handle for the ACS stream.

eventClass - This is a tag with the value **CSTACONFIRMATION**, which identifies this message as an ACS confirmation event.

eventType - This is a tag with the value **CSTA_QUERY_CALL_MONITOR_CONF**, which identifies this message as an **CSTAQueryCallMonitorConfEvent**.

invokeID - This parameter specifies the requested instance of the function. It is used to match a specific function request with its confirmation events.

callMonitor - This parameter indicates whether or not (TRUE or FALSE) the ACS stream has call/call monitoring privilege.

CSTA Event Data Types

This section defines all the event data types which are used with the CSTA functions and messages and may repeat data types already shown in the CSTA Control Functions. Refer to the specific commands for any operational differences in these data types. The complete set of CSTA data types is given in ACS Data Types on page 114. The CSTA data types are type defined in the **CSTA.H** header file.

An application always receives a generic *CSTAEvent_t* event structure. This structure contains an *ACSEventHeader_t* structure which contains information common to all events. This common information includes:

- *acsHandle:* Specifies the ACS stream the event arrived on.
- *eventClass:* Identifies the event as an ACS confirmation, ACS unsolicited, CSTA confirmation, or CSTA unsolicited event.
- *eventType*: Identifies the specific type of message (MakeCall, confirmation event, HoldCall event, etc.)
- *privateData:* Private data defined by the specified driver vendor.

The *CSTAEvent_t* structure then consists of a union of the four possible *eventClass* types; ACS confirmation, ACS unsolicited, CSTA confirmation or CSTA unsolicited event. Each *eventClass* type itself consists of a union of all the possible *eventTypes* for that class. Each eventClass may contain common information such as *invokeID* and *monitorCrossRefID.*

```
/* CSTA Control Services Header File <CSTA.H> */

   #include <acs.h>

// defines for CSTA event classes

   #define   CSTAREQUEST    3
   #define   CSTAUNSOLICITED    4
   #define   CSTACONFIRMATION    5
   #define   CSTAEVENTREPORT    6

typedef struct {
   InvokeID_t    invokeID;
   union
   {
          CSTARouteRequestEvent_t      routeRequest;
          CSTARouteRequestExtEvent_t   routeRequestExt;
          CSTAReRouteRequest_t         reRouteRequest;
          CSTAEscapeSvcReqEvent_t      escapeSvcReqeust;
          CSTASysStatReqEvent_t        sysStatRequest;
   } u;
} CSTARequestEvent;

typedef struct {
   union
   {
          CSTARouteRegisterAbortEvent_tregisterAbort;
          CSTARouteUsedEvent_t         routeUsed;
          CSTARouteUsedExtEvent_t      routeUsedExt;
          CSTARouteEndEvent_t          routeEnd;
          CSTAPrivateEvent_t           privateEvent;
          CSTASysStatEvent_t           sysStat;
          CSTASysStatEndedEvent_t      sysStatEnded;
          }u;
   } CSTAEventReport;

typedef struct {
    CSTAMonitorCrossRefID_t     monitorCrossRefId;
    union
    {
          CSTACallClearedEvent_t       callCleared;
          CSTAConferencedEvent_t       conferenced;
          CSTAConnectionClearedEvent_t connectionCleared;
          CSTADeliveredEvent_t         delivered;
          CSTADivertedEvent_t          diverted;
          CSTAEstablishedEvent_t       established;
          CSTAFailedEvent_t            failed;
          CSTAHeldEvent_t              held;
          CSTANetworkReachedEvent_t    networkReached;
          CSTAOriginatedEvent_t        originated;
```

```
            CSTAQueuedEvent_t           queued;
            CSTARetrievedEvent_t        retrieved;
            CSTAServiceInitiatedEvent_t serviceInitiated;
            CSTATransferredEvent_t      transferred;
            CSTACallInformationEvent_t  callInformation;
            CSTADoNotDisturbEvent_t     doNotDisturb;
            CSTAForwardingEvent_t       forwarding;
            CSTAMessageWaitingEvent_t   messageWaiting;
            CSTALoggedOnEvent_t         loggedOn;
            CSTALoggedOffEvent_t        loggedOff;
            CSTANotReadyEvent_t         notReady;
            CSTAReadyEvent_t            ready;
            CSTAWorkNotReadyEvent_t     workNotReady;
            CSTAWorkReadyEvent_t        workReady;
            CSTABackInServiceEvent_t    backInService;
            CSTAOutOfServiceEvent_t     outOfService;
            CSTAPrivateStatusEvent_t    privateStatus;
            CSTAMonitorEndedEvent_t     monitorEnded;
        } u;
    } CSTAUnsolicitedEvent;

        typedef struct
        {
        InvokeID_t    invokeID;
        union
        {
            CSTAAlternateCallConfEvent_t  alternateCall;
            CSTAAnswerCallConfEvent_t     answerCall;
            CSTACallCompletionConfEvent_t callCompletion;
            CSTAClearCallConfEvent_t      clearCall;
            CSTAClearConnectionConfEvent_t clearConnection;
            CSTAConferenceCallConfEvent_t conferenceCall;
            CSTAConsultationCallConfEvent_t consultationCall;
            CSTADeflectCallConfEvent_t    deflectCall;
            CSTAPickupCallConfEvent_t     pickupCall;
            CSTAGroupPickupCallConfEvent_t groupPickupCall;
            CSTAHoldCallConfEvent_t       holdCall;
            CSTAMakeCallConfEvent_t       makeCall;
            CSTAMakePredictiveCallConfEvent_t makePredictiveCall;
            CSTAQueryMwiConfEvent_t       queryMwi;
            CSTAQueryDndConfEvent_t       queryDnd;
            CSTAQueryFwdConfEvent_t       queryFwd;
            CSTAQueryAgentStateConfEvent_t queryAgentState;
            CSTAQueryLastNumberConfEvent_t queryLastNumber;
            CSTAQueryDeviceInfoConfEvent_t queryDeviceInfo;
            CSTAReconnectCallConfEvent_t  reconnectCall;
            CSTARetrieveCallConfEvent_t   retrieveCall;
            CSTASetMwiConfEvent_t         setMwi;
            CSTASetDndConfEvent_t         setDnd;
            CSTASetFwdConfEvent_t         setFwd;
```

```
            CSTASetAgentStateConfEvent_t  setAgentState;
            CSTATransferCallConfEvent_t   ransferCall;
            CSTAUniversalFailureConfEvent_t universalFailure;
            CSTAMonitorConfEvent_t        monitorStart;
            CSTAChangeMonitorFilterConfEvent_t changeMonitorFilter;
            CSTAMonitorStopConfEvent_t    monitorStop;
            CSTASnapshotDeviceConfEvent_t snapshotDevice;
            CSTASnapshotCallConfEvent_t   snapshotCall;
            CSTARouteRegisterReqConfEvent_t routeRegister;
            CSTARouteRegisterCancelConfEvent_t routeCancel;
            CSTAEscapeSvcConfEvent_t      escapeService;
            CSTASysStatReqConfEvent_t     sysStatReq;
            CSTASySStatStartConfEvent_t   sysStatStart;
            CSTASysStatStopConfEvent_t    sysStatStop;
            CSTAChangeSysStatFilterConfEvent_t changeSysStatFilter;
            CSTAGetAPICapsConfEvent_t     getAPICaps;
            CSTAGetDeviceListConfEvent_t  getDeviceList;
            CSTAQueryCallMonitorConfEvent_t queryCallMonitor;
            } u;
  } CSTAConfirmationEvent;

#define CSTA_MAX_HEAP 1024

typedef struct
     {
ACSEventHeader_t eventHeader;
union
{
     ACSUnsolicitedEvent   acsUnsolicited;
     ACSConfirmationEvent  acsConfirmation;
     CSTARequestEvent      cstaRequest;
     CSTAUnsolicitedEvent  cstaUnsolicited;
     CSTAConfirmationEvent  cstaConfirmation;
     } event;
char   heap[CSTA_MAX_HEAP];
} CSTAEvent_t
```

Chapter 4: CSTA Service Groups supported by the TSAPI Service

This chapter describes the CSTA Services Groups that the Application Enablement Services TSAPI Service supports. It includes the following topics.

Supported Services and Service Groups

The AE Services TSAPI Service supports the service groups defined in Table 6. Services that are not supported are listed in Table 7.

Table 6: Supported CSTA Services for Communication Manager

Service Group	Service Group Definition	Supported Service(s)
Call Control	The services in this group enable a telephony client application to control a call or connection on Communication Manager. Typical uses of these services are: placing calls from a device controlling a connection for a single call.	Alternate Call Answer Call Clear Call Clear Connection Conference Call Consultation Call Consultation-Direct-Agent Call (private) Consultation Supervisor-Assist Call (private) Deflect Call Hold Call Make Call Make Direct-Agent Call (private) Make Predictive Call Make Supervisor-Assist Call (private) Pickup Call Reconnect Call Retrieve Call Selective Listening Hold (private) Selective Listening Retrieve (private V5) Send DTMF Tone (private) Single Step Conference (private) Transfer Call Single Step Transfer (private)

Table 6: Supported CSTA Services for Communication Manager (continued)

Service Group	Service Group Definition	Supported Service(s)
Set Feature	The services in this group allow a client application to set switch-controlled features or values on a Communication Manager device.	Set Advice Of Charge (private) Set Agent State Set Bill Rate (private) Set Do Not Disturb Set Forwarding Set Message Waiting Indicator
Query	The services in this group allow a client to query device features and static attributes of a Communication Manager device.	Query ACD Split (private) Query Agent Login (private) Query Agent Measurements (private) Query Agent State Query Call Classifier (private) Query Device Info Query Device Name Query Do Not Disturb Query Forwarding Query Message Waiting Indicator Query Split/Skill Measurements (private) Query Time of Day (private) Query Trunk Group (private) Query Trunk Group Measurements (private) Query Station Status (private) Query Universal Call ID (private) Query VDN Measurements (private)
Snapshot	The services in this group allow a client application to take a snapshot of a call or device on a Communication Manager server.	Snapshot Call Snapshot Device
Monitor	The services in this group allow a client application to request and cancel the reporting of events that cause a change in the state of a Communication Manager object.	Change Monitor Filter Monitor Call Monitor Calls Via Device Monitor Device Monitor Ended Event Monitor Stop on Call (private) Monitor Stop

Table 6: Supported CSTA Services for Communication Manager (continued)

Service Group	Service Group Definition	Supported Service(s)
Event Report	The services in this group provide a client application with the reports of events that cause a change in the state of a call, a connection, or a device.	Call Event Reports: • Call Cleared • Charge Advice (private V5) • Connection Cleared • Conferenced • Delivered • Diverted • Entered Digits (private) • Established • Failed • Held • Network Reached • Originated • Queued • Retrieved • Service Initiated • Transferred Agent State Event Reports: • Logged On • Logged Off Feature Event Reports: • Do Not Disturb • Forwarding
Routing	The services in this group allow Communication Manager to request and receive routing instructions for a call from a client application.	Route End Event Route End Service Route Register Abort Event Route Register Cancel Service Route Register Service Route Request Service Route Select Service Route Used Event
Escape	The services in this group allow an application to request a private service that is not defined by the CSTA Standard.	Escape Service Private Event Private Status Event

Table 6: Supported CSTA Services for Communication Manager (continued)

Service Group	Service Group Definition	Supported Service(s)
Maintenance	The services in this group allow an application to request (1) device status maintenance events that provide status information for device objects, and (2) bi-directional system status maintenance services that provide information on the overall status of the system.	None
System Status	The services in this group allow an application to request system status information from the TSAPI Service.	System Status Request System Status Start System Status Stop Change System Status Filter System Status Event

Table 7: Unsupported CSTA Services

Service Group	Unsupported Service(s) or Event Report(s)
Call Control	Group Pickup Call
Set Feature	None
Query	Query Last Number
Snapshot	None
Monitor	None
Event Reports	Call Event Reports: None Agent State Event Reports: Not Ready Event Ready Event Work Not Ready Event Work Ready Event Feature Event Reports: Call Info Event Message Waiting Event
Routing	Re-Route Event
Escape	Send Private Event

Table 7: Unsupported CSTA Services (continued)

Service Group	Unsupported Service(s) or Event Report(s)
Maintenance	Back in Service Event Out of Service Event
System Status	System Status Request Event System Status Ended Event System Status Event Send

CSTA Objects

Figure 4 illustrates the three types of CSTA objects: Device, Call, and Connection.

Figure 4: CSTA Objects: Device, Call and Connection

CSTA Object: Device

The term device refers to both physical devices (stations, trunks, and so on) and logical devices (VDNs or ACD splits) that are controlled by the switch. Each device is characterized by a set of attributes. These attributes define the manner in which an application may observe and manipulate a device. The set of device attributes consists of:

- Device Type, for more information, see Device Type on page 144
- Device Class, for more information, see Device Class on page 145
- Device Identifier, for more information, see Device Identifier on page 148

Device Type

Table 8 defines the most commonly used Communication Manager devices and their types:

Table 8: CSTA Device Type Definitions

CSTA Type	Definition	Communication Manager Object
Station	A traditional telephone device or an AWOH station extension (for phantom calls).[1] A station is a physical unit of one or more buttons and one or more lines.	Station or extension on Communication Manager.
ACD Group	A mechanism that distributes calls within a switch.	VDN, ACD split, or hunt group in Communication Manager.
Trunk	A device used to access other switches.	Trunk
Trunk Group	A group of trunks accessed using a single identifier.	Trunk group
Other	A type of device not defined by CSTA.	Announcement, CTI (ASAI), modem pool, etc.

1. A call can be originated from an AWOH station or some group extensions (i.e., a plain [non-ACD] hunt group). This is termed a phantom call. Most calls that can be requested for a physical extension can also be requested for an AWOH station and the associated event will also be received. If the call is made on behalf of a group extension, this may not apply. For more information about the phantom call switch feature, refer to the *Avaya MultiVantage Application Enablement Services ASAI Technical Reference*.

CSTA Device Types that the TSAPI Service does not support

CSTA defines device types that the TSAPI Service does not use.

- ACD Group
- button
- button group
- line
- line group
- operator
- operator group
- station group

Device Class

Different classes of devices can be observed and manipulated within the TSAPI Service CSTA environment. Common Communication Manager CSTA Device Classes include: voice and other. The TSAPI Service does not support service requests for the CSTA data and image classes. The TSAPI Service may return the data class in response to a query.

Device History

The DeviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the DeviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

Important:
Device History cannot be guaranteed for events that happened before monitoring started. Note that the cause value should be EC_NETWORKSIGNAL if an ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Format of the Device History parameter

The Device History parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call. Each entry consists of:

entry	description
olddeviceID (M) DeviceID	the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following: ● any device identifier format. ● "Not Known" - indicates that the device identifier associated with this entry in the DeviceHistory list cannot be provided. ● "Restricted" - indicates that the device associated with this entry in the DeviceHistory list cannot be provided due to regulatory and/or privacy reasons. ● "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID is not provided with this list entry. ● "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID is not provided with this list entry.
Cause (O) EventCause	The reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
OldconnectionID (O) ConnectionID	The CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

The value of the deviceHistoryCount parameter

For AE Services the value of deviceHistoryCount is 1.

The deviceHistoryCount parameter is used for storing the number of entries in the DeviceHistory parameter. AE Services supports only one entry, with a limit of 1, in the DeviceHistory parameter. When the limit of 1 is reached, the new value replaces the old value (unless specifically stated otherwise in this document).

Merging calls - DeviceHistory

The source for DeviceHistory data is always the Primary Old Call when merging calls.

Interactions:

GetAPICapps will return deviceHistoryCount, and for AE Services the value 1 (one) will always be returned.

Device Identifier

Each device that can be observed and manipulated needs to be referenced across the CSTA Service boundary. Devices are identified using one or both of the following types of identifiers:

Static Device Identifier

A static device identifier is stable over time and remains both constant and unique between calls.

The static device identifier is known by both the TSAPI application and the Communication Manager Server. Communication Manager internal extensions are static device identifiers. These include extensions that uniquely identify any Communication Manager devices such as stations or AWOH station extensions (for phantom calls), ACD splits, VDNs, and logical agent login IDs. Valid phone numbers for endpoints external to Communication Manager Server are also static device identifiers.

Note:
If applicable, access and authorization codes can be specified with the static device identifier for the called device parameter of the Make Call Service.

The presence of a static device ID in an event does not necessarily mean that the device is directly connected to the switch.

Note:

If the called device specified in a CSTA Make Call Service request is not an internal endpoint, the device identifier reported in the event reports for that device on that call may not be the same. The called device specified in the CSTA Make Call Service is a dialing digit sequence and it may not represent a true device identifier. For example, the trunk access code can be specified as part of the dialing digits in the called device parameter of a CSTA Make Call Service request. However, the trunk access code will not be part of the device identifier of the called device in the event reports of that call. In a DCS (Distributed Communications System) or SDN (Software Defined Network) environment, even if a true device identifier (such as one with no trunk access code in the called device parameter) of an external endpoint is specified for the called device in a CSTA Make Call Service request, Communication Manager may not use the same device identifier in the event reports for the called device.

Dynamic Device Identifier

When a call is connected through a trunk with an unknown device identifier, a dynamic trunk identifier is created for the purpose of identifying the external endpoint. This identifier is not like a static device identifier that an application can store in a database for later use. An off-PBX endpoint without a known static identifier has a trunk identifier.

Note:

An off-PBX endpoint of an ISDN call may have a known static identifier

Bear in mind that a trunk identifier does not identify the actual trunk or trunk group to which the endpoint is connected. The actual trunk and trunk group information, if available, is provided in the Private Data.

To manipulate and monitor calls that cross a Communication Manager trunk interface, an application needs to use the trunk identifier. The TSAPI Service preserves trunk identifiers across conference and transfer operations. The TSAPI Service may use different dynamic identifiers to represent endpoints connected to the same actual trunk at different times. A trunk identifier is meaningful to an application only for the duration of a call and should not be retained and used at a later time, for example, a phone number or a station extension. A call identifier and a trunk identifier can comprise a connection identifier. A trunk identifier has a prefix ’T’ and a ’#’ within its identifier (for example, T538#1, T4893#2).

Device ID Type

If an application opens an ACS stream with Private Data Version 5 and later, the TSAPI Service supports CSTA DeviceIDType_t based on information from the switch, network, or internal information.

- IMPLICIT_PUBLIC (20) - There is no actual numbering and addressing information about this endpoint received from the network or switch. However, from the number of digits (7 or more digits) of the device identifier associated with this endpoint, it may be a public number. Prefix or escape digits may be present.
- EXPLICIT_PUBLIC_UNKNOWN (30) - There are two cases for this type:
  - There is no actual numbering and addressing information about this endpoint received from the network or switch. The network or switch did not provide any actual numbering or addressing information about this endpoint. The device identifier is also unknown for this endpoint. An external endpoint without a known device identifier is most likely to have this type.
  - The numbering and addressing information are provided by the ISDN interface from the network and the Communication Manager Server that the call is connected to, but the network and switch have no knowledge about the number (whether it is international, national, or local) or the endpoint. Prefix or escape digits may be present.
- EXPLICIT_PUBLIC_INTERNATIONAL (31) - This endpoint has an international number. The numbering plan and addressing type information are provided by the ISDN interface from the network and the Communication Manager server the call is connected to. Prefix or escape digits are not included.
- EXPLICIT_PUBLIC_NATIONAL (32) - This endpoint has a national number. The numbering plan and addressing type information are provided by the ISDN interface from the network and the Communication Manager server the call is connected to. Prefix or escape digits are not included.
- EXPLICIT_PUBLIC_NETWORK_SPECIFIC (33) - This endpoint has a network specific number. The numbering plan and addressing type information are provided by the ISDN interface from the network and the Communication Manager server the call is connected to. The type of network specific number is used to indicate the administration/service number specific to the serving network, (e.g., used to access an operator).
- EXPLICIT_PUBLIC_SUBSCRIBER (34) - This endpoint has a network specific number. The numbering plan and addressing type information are provided by the ISDN interface from the network and the Communication Manager Server the call is connected to. Prefix or escape digits are not included.
- EXPLICIT_PUBLIC_ABBREVIATED (35) - This endpoint has an abbreviated number. The numbering and addressing information are provided by the ISDN interface from the network and the Communication Manager Server the call is connected to.

- IMPLICIT_PRIVATE (40) - There is no actual numbering plan and addressing type information about this endpoint received from the network or switch. However, from the number of digits (6 or less digits) of the device identifier associated with this endpoint, it may be a private number. Prefix or escape digits may be present. An internal endpoint or an external endpoint across the DCS or private network may have this type. Note that it is not unusual for an internal endpoint's type changing from IMPLICIT_PRIVATE to EXPLICIT_PRIVATE_LOCAL_NUMBER when more information about the endpoint is received from the switch.
- EXPLICIT_PRIVATE_UNKNOWN (50) - This endpoint has a private numbering plan and the addressing type is unknown. An endpoint is unlikely to have this device ID type.
- EXPLICIT_PRIVATE_LEVEL3_REGIONAL_NUMBER (51) - This endpoint has a private numbering plan and its addressing type is level 3 regional. An endpoint is unlikely to have this device ID type.
- EXPLICIT_PRIVATE_LEVEL2_REGIONAL_NUMBER (52) - This endpoint has a private numbering plan and its addressing type is level 2 regional. An endpoint is unlikely to have this device ID type.
- EXPLICIT_PRIVATE_LEVEL1_REGIONAL_NUMBER (53) - This endpoint has a private numbering plan and its addressing type is level 1 regional. An endpoint is unlikely to have this device ID type.
- EXPLICIT_PRIVATE_PTN_SPECIFIC_NUMBER (54) - This endpoint has a private numbering plan and its addressing type is PTN specific. An endpoint is unlikely to have this device ID type.
- EXPLICIT_PRIVATE_LOCAL_NUMBER (55) - There are two cases for this type:
  - There is no actual numbering plan and addressing type information about this endpoint received from the switch or network. However, this endpoint has a device identifier and its type is identified by the TSAPI Service as a local number or a local endpoint to Communication Manager Server.

    A local endpoint is one that is directly connected to Communication Manager Server that the TSAPI Service is connected to. An endpoint that is not directly connected to a Communication Manager Server and the TSAPI Service, but can be accessed through the DCS or private network Communication Manager Server and the TSAPI Service is not a local endpoint. A TSAPI Service local endpoint normally has a type of either EXPLICIT_PRIVATE_LOCAL_NUMBER or IMPLICIT_PRIVATE. Note that it is not unusual for an endpoint's type to change from IMPLICIT_PRIVATE to EXPLICIT_PRIVATE_LOCAL_NUMBER when more information about the endpoint is received from the switch. An internal endpoint is most likely to have this device ID type in this case.
  - This endpoint has a private numbering plan and its addressing type is local number. An endpoint is unlikely to have this device ID type with this case.
- EXPLICIT_PRIVATE_ABBREVIATED (56) - This endpoint has a private numbering plan and its addressing type is abbreviated. An endpoint is unlikely to have this device ID type.

- OTHER_PLAN (60) - This endpoint has a type “none of the above.” An endpoint is unlikely to have this type.
- TRUNK_GROUP_IDENTIFIER (71) - This type is not used by the TSAPI Service.

Device Identifier Syntax

```
typedef char            DeviceID_t[64];

typedef enum DeviceIDType_t {
DEVICE_IDENTIFIER = 0,
IMPLICIT_PUBLIC = 20,
EXPLICIT_PUBLIC_UNKNOWN = 30,
EXPLICIT_PUBLIC_INTERNATIONAL = 31,
EXPLICIT_PUBLIC_NATIONAL = 32,
EXPLICIT_PUBLIC_NETWORK_SPECIFIC = 33,
EXPLICIT_PUBLIC_SUBSCRIBER = 34,
EXPLICIT_PUBLIC_ABBREVIATED = 35,
IMPLICIT_PRIVATE = 40,
EXPLICIT_PRIVATE_UNKNOWN = 50,
EXPLICIT_PRIVATE_LEVEL3_REGIONAL_NUMBER = 51,
EXPLICIT_PRIVATE_LEVEL2_REGIONAL_NUMBER = 52,
EXPLICIT_PRIVATE_LEVEL1_REGIONAL_NUMBER = 53,
EXPLICIT_PRIVATE_PTN_SPECIFIC_NUMBER = 54,
EXPLICIT_PRIVATE_LOCAL_NUMBER = 55,
EXPLICIT_PRIVATE_ABBREVIATED = 56,
OTHER_PLAN = 60,
TRUNK_IDENTIFIER=70,
TRUNK_GROUP_IDENTIFIER=71
} DeviceIDType_t;

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

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

typedef ExtendedDeviceID_t CallingDeviceID_t;

typedef ExtendedDeviceID_t CalledDeviceID_t;

typedef ExtendedDeviceID_t SubjectDeviceID_t;

typedef ExtendedDeviceID_t RedirectionDeviceID_t;
```

The TSAPI Service Call object

Applications can use TSAPI to control and monitor Call behavior, including establishment and release. There are two types of call attributes:

- Identifier - see Call Identifier (callID) on page 154
- State - Call State on page 154

Call Identifier (callID)

When a call is initiated, Communication Manager allocates a unique Call Identifier (callID). Before a call terminates, it may progress through many different states involving a variety of devices. Although the call identifier may change (as with transfer and conference, for example), its status as a CSTA object remains the same. A callID first becomes visible to an application when it appears in an event report or confirmation event. The allocation of a callID is always reported. Each callID is specified in a connection identifier parameter.

Note:
The TSAPI interface passes callID parameters within connectionID parameters.

Call Identifier Syntax

```
typedef struct ConnectionID_t {
            long        callID;     // always specified in a
                                    // connectionID
            DeviceID_t  deviceID;   // set to 0, when only callID
                                    // is interested
ConnectionID_Device_t  devIDType;   // STATIC_ID or DYNAMIC_ID
            } ConnectionID_t;
```

Call State

A “call state” is a descriptor (initiated, queued, etc.) that characterizes the state of a call. Even though a call may assume several different states throughout its duration, it can only be in a single state at any given time. The set of connection states comprises all of the possible states a call may assume. Call state is returned by the Snapshot Device Service for devices that have calls.

The TSAPI Service Connection object

A “connection,” as defined by CSTA, is a relationship that exists between a call and a device. Many API Services (Hold Call Service, Retrieve Call Service, and Clear Call Service, for example) observe and manipulate connections. Connections have the following attributes:

- Identifier - for more information, see Connection Identifier (connectionID) on page 155
- State - for more information, see Connection State on page 156

Connection Identifier (connectionID)

A connectionID is a combination of Call Identifier (callID) and Device Identifier (deviceID). The connectionID is unique within a Communication Manager server. An application can not use a connectionID until it has received it from the TSAPI Service. This rule prevents an application from fabricating a connectionID.

A connectionID always contains a callID value. A TSAPI Service connectionID may contain a static or dynamic (for Trunk ID) device identifier. If the callID is the only value that is present, the deviceID is set to 0 (with DYNAMIC_ID). The callID of a connectionID assigned to an endpoint on a call may change when the call is transferred or conferenced, but the deviceID of the connectionID assigned to an endpoint will not change when the call is transferred or conferenced.

For a call, there are as many Connection Identifiers as there are devices on the call. For a device, there are as many Connection Identifiers as there are calls at that device.

Connection Identifier Conflict : A device may connect to a call twice. This can happen for external endpoints with the same calling number from an ISDN network or from an internal device with different line appearances connected to the same call. In these rare cases, the TSAPI Service resolves the device identifier conflict in the connection identifiers by replacing one of the device identifiers with a trunk identifier when two calls that have the same device (this is not the device conferencing the call) on them are merged by a call conference or transfer operation.

Note:
The connection identifier of a device on a call can change in this case.

Connection Identifier Syntax

```
typedef char            DeviceID_t[64];

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

typedef struct ConnectionID_t {
            long                    callID;
            DeviceID_t              deviceID;
            ConnectionID_Device_t   devIDType;
} ConnectionID_t;
```

Connection State

A connection state is a descriptor (initiated, queued, etc.) that characterizes the state of a single CSTA connection. Connection states are reported by Snapshots taken of calls or devices. Changes in connection states are reported as event reports by Monitor Services.

Figure 5 illustrates a connection state model that shows typical connection state changes. This connection state model derives from the CSTA connection state model. It provides an abstract view of various call state transitions that can occur when a call is either initiated from, or delivered to, a device. Note that this model does not include all the possible states that may result from interactions with Communication Manager features, and it does not represent a complete programming model for the call state/event report/connection state relationship. The Communication Manager Server also incorporates state transitions that may not be shown.

Note:

It is strongly recommended that applications be event driven. Being state driven, rather than event driven, may result in an unexpected state transition that the program has not anticipated. This often occurs because some party on the call invokes a Communication Manager feature that interacts with the call in a way that is not part of a typical call flow. The diagram that follows captures only typical call state transitions. Communication Manager has a large number of specialized features that interact with calls in many ways.

In Figure 5, circles represent connection states. Arrows are indicate transitions between states. A transition from one connection state to another results in the generation of an event report. The various connection states are defined Table 9.

Figure 5: AE Services TSAPI Service Sample Connection State Model

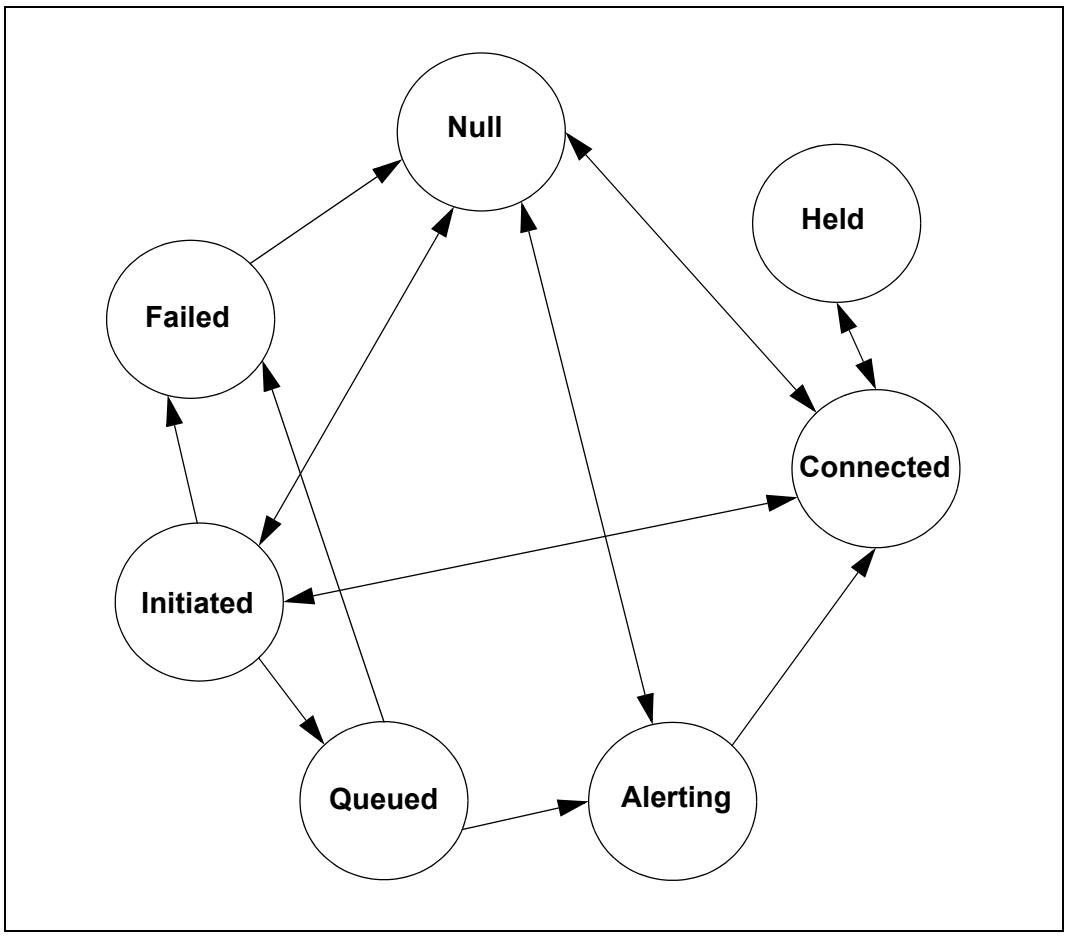


Table 9: TSAPI Service Connection State Definitions

Definition	Description
Null	No relationship exists between the call and device; a device does not participate in a call.
Initiated	A device is requesting service. Usually, this results in the creation of a call. Often, this is when a station receives a dial tone and begins to dial.
Alerting	A device is alerting (ringing). A call is attempting to become connected to a device. The term “active” is also used to indicate an alerting (or connected) state.
Connected	A device is actively participating in a call, either logically or physically (that is, not Held). The term “active” is also used to indicate a connected (or alerting) state.
Held	A device inactively participates in a call. That is, the device participates logically but not physically.
Queued	Normal state progression has been stalled. Generally, either a device is trying to establish a connection with a call or a call is trying to establish a connection with a device.
Failed	Normal state progression has been aborted. Generally, either a device is trying to establish a connection with a call or a call is trying to establish a connection with a device. A Failed state can result from a failure to connect to the calling device (origin) or to the called device (destination). A Failed state can also be caused by a failure to create the call or other factors.
Unknown	A device participates in a call, but its state is not known.
Bridged	This is a Communication Manager Server private local connection state that is not defined by CSTA. This state indicates that a call is present at a bridged, simulated bridged, button TEG, or POOL appearance, and the call is neither ringing nor connected at the station. The bridged connection state is reported in the private data of a Snapshot Device Confirmation Event and it has a CSTA null (CS_NULL) state. Since this is the only time TSAPI Service returns CS_NULL, a device with the null state in the Snapshot Device Confirmation Event is bridged.
	A device with the bridged state can join the call by either manually answering the call or the cstaAnswerCall Service. Once a bridged device is connected to a call, its state becomes connected. After a bridged device becomes connected, it can drop from the call and become bridged again, if there are other endpoints still on the call.
	Manual drop of a bridged line appearance (from the connected state) from a call will not cause a Connection Cleared Event.

Connection State Syntax

```
typedef enum LocalConnectionState_t {
    CS_NONE = -1,
    CS_NULL = 0,
    CS_INITIATE = 1,
    CS_ALERTING = 2,
    CS_CONNECT = 3,
    CS_HOLD = 4,
    CS_QUEUED = 5,
    CS_FAIL = 6
} LocalConnectionState_t;
```

CSTAUniversalFailureConfEvent

The CSTA universal failure confirmation event provides a generic negative response from the server/switch for a previously requested service. The CSTAUniversalFailureConfEvent will be sent in place of any confirmation event described in each service function description when the requested function fails.The confirmation events defined for each service function are only sent when that function completes successfully.

For a listing of common CSTA error messages, see Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

Chapter 5: Avaya TSAPI Service Private Data

This chapter describes the private data features that the Avaya MultiVantage Application Enablement Services (AE Services) TSAPI Service provides.

What is private data?

Private data is the means for both extending the functionality of any defined CSTA service and for providing additional functionality altogether. The TSAPI Service uses the “private data” mechanism to provide applications with access to special features of Avaya Communication Manager.

Private data may be defined for each CSTA service request, CSTA confirmation event and CSTA unsolicited event. In concrete terms, Avaya is free to privately define a specific 'extension message' to be carried along with any CSTA message.

The set of fields in a CSTA message is called a protocol data unit, or PDU. So each CSTA message defines a PDU. The set of fields that accompany a particular CSTA PDU, representing the extended functionality that Avaya provides for that CSTA PDU, defines a the private Avaya protocol data unit or private PDU corresponding to that CSTA PDU.

The CSTA PDUs, as supported by TSAPI service, are defined by ECMA-180 and are unchanging in content. The way Avaya extends the functionality of a CSTA event is by promising to provide an enhanced private PDU to accompany that CSTA PDU; for example, when sending the CSTA PDU for the Delivered Event (CSTADeliveredEvent_t), Avaya can provide ISDN User-To-User Information (UUI) and other data in a private PDU called ATTDeliveredEvent_t (the ATT is present for historical reasons).

What is a private data version

Private data allows a PBX or switch manufacturer to extend the base set of TSAPI capabilities. Over time, a PBX manufacturer may choose to further enhance the capabilities that are available using private data.

A private data version defines a fixed set of these capabilities. More specifically, it defines a set of escape services and private event parameters for CSTA events. This lets the application developer know exactly which services and private data items are available. Having the ability to negotiate a specific private data version ensures that an application written for an earlier release of AE Services will continue to operate with newer releases.

Each private data version is designated by a number (for example, private data version 8 or PDV8). With the latest product release of the Application Enablement Services (AE Services 4.1) an application may ask the TSAPI service to provide data defined for private data versions 2 through 8. Newer features and content are provided with higher numbered private data versions. Private data versioning is inclusive. If you negotiate private data version 8, you have access to all the capabilities of previous private data versions.

It is important to note, however, that the confirmation event to a request will always be returned in the latest format available within the private data version negotiated, even if the request is sent in the format of a previous data version. For example, if an application negotiates private data version 8 for the stream and sends a request using a private data version 4 format, then the confirmation event will be returned in the latest format available for that event up to and including private data version 8. If the application, in this example, needed to ensure that confirmation event was returned in a format no later than version 4, then it should have initially negotiated private data version 4 for the stream, not version 8.

See Table 10 for a summary history of private data versions.

Table 10: Summary of Private Data Versions

Product	Supported private data versions
Avaya Computer Telephony	PDV 2 through 6
Application Enablement Services 3.0	PDV 2 through 6
Application Enablement Services 3.1	PDV 2 through 7
Application Enablement Services 4.0	PDV 2 through 7
Application Enablement Services 4.1	PDV 2 through 8

Linking your application to the private data functions

AE Services defines the mechanism for private data in a dynamically linked or shared library file, which contains private data encoding and decoding functions. For Windows-based clients, this file is ATTPRIV32.DLL. For Unix-based clients, this file is libattpriv.so. If your application uses private data, you must link to this file.

Summary of TSAPI Service Private Data

Table 11 summarizes private data features provided by the AE Services TSAPI Service. The features listed as PDV 8 in the right column are new features for Release 4.1 of the TSAPI Service. For more information previous version of private data, see Appendix B: Summary of Private data support on page 811.

Table 11: Private Data Summary

Private Data Feature	Initial Private Data Version
Single Step Transfer Call	PDV 8
Calling Device in Failed Event	PDV 8
Enhanced Monitor Calls via Device **Note:** To get this PDV 7 capability you must upgrade to either AE Services 3.1.4 (Service Pack Release) or AE Services 4.1.	PDV7
Network Call Redirection for Routing	PDV 7
Redirecting Number Information Element (presented through DeviceHistory)	PDV 7
Query Device Name for Attendants	PDV 7
Increased Aux Reason Codes	PDV 7
Enhanced GetAPICaps Version	PDV 7
Pending Work Mode and Pending Reason Code in Set Agent State and Query Agent State	PDV 6
Trunk Group and Trunk Member Information in Delivered Event and Established Event regardless of whether Calling Party is Available	PDV 6
Trunk Group Information in Route Request Events regardless of whether Calling Party is Available	PDV 6
Trunk Group Information for Every Party in Transferred Events and Conferenced Events	PDV 6
User-to-User Info (UUI) is increased from 32 to 96 bytes	PDV 6
Support Detailed DeviceIDType_t in Events	PDV 5
Set Bill Rate	PDV 5
Flexible Billing in Delivered Event, Established Event, and Route Request	PDV 5
Call Originator Type in Delivered Event, Established Event, and Route Request	PDV 5

Table 11: Private Data Summary (continued)

Private Data Feature	Initial Private Data Version
Selective Listening Hold	PDV 5
Selective Listening Retrieve	PDV 5
Set Advice of Charge	PDV 5
Charge Advice Event	PDV 5
Reason Code in Set Agent State, Query Agent State, and Logout Event	PDV 5
27-Character Display Query Device Name Confirmation	PDV 5
Unicode Device ID in Events	PDV 5
Trunk Group and Trunk Member Information in Network Reached Event	PDV 5
Universal Call ID (UCID) in Events	PDV 5
Single Step Conference	PDV 5
Distributing Device in Conferenced, Delivered, Established, and Transferred Events	PDV 4
Private Capabilities in cstaGetAPICaps Confirmation Private Data	PDV 4
Deflect Call	PDV 3
Pickup Call	PDV 3
Originated Event Report	PDV 3
Agent Logon Event Report	PDV 3
Reason for Redirection in Alerting Event Report	PDV 3
Agent, Split, Trunk, VDN Measurements Query	PDV 3
Device Name Query	PDV 3
Send DTMF Tone	PDV 3
Priority, Direct Agent, Supervisor Assist Calling	PDV 2
Enhanced Call Classification	PDV 2
Trunk, Classifier Queries	PDV 2
LAI in Events	PDV 2
Launching Predictive Calls from Split	PDV 2
Application Integration with Expert Agent Selection	PDV 2

Table 11: Private Data Summary (continued)

Private Data Feature	Initial Private Data Version
User-to-User Info (Reporting and Sending)	PDV 2
Multiple Notification Monitors (two on ACD/VDN)	PDV 2
Launching Predictive Calls from VDN	PDV2
Multiple Outstanding Route Requests for One Call	PDV 2
Answering Machine Detection	PDV 2
Established Event for Non-ISDN Trunks	PDV 2
Provided Prompter Digits on Route Select	PDV 2
Requested Digit Selection	PDV 2
VDN Return Destination (Serial Calling)	PDV 2
Prompted Digits in Delivered Events	PDV 1

Private Data Version 8 Features

AE Services TSAPI Service, Release 4.1, provides the following new features for Private Data Version 8.

- Single Step Transfer Call - see Single Step Transfer Call on page 168.
- Calling Device in Failed Event - see Calling Device in Failed Event on page 168.
- New Get API Capabilities confirmation event - see CSTA Get API Capabilities confirmation structures for Private Data Version 8 on page 171.
- A new private data parameter, flowPredictiveCallEvents. for the CSTAMonitorCallsViaDevice service. For more information, see Monitor Calls Via Device Service on page 462.

Single Step Transfer Call

The Single Step Transfer Call service transfers an existing connection to another device, and it performs this transfer in a single step. This means that the device transferring the call does not have to place the existing call on hold before issuing the Single Step Transfer Call service. For a service description, see Single Step Transfer Call (Private Data Version 8 and later) on page 327.

Calling Device in Failed Event

The Failed Event includes the Calling Device, if available.

```
typedef struct ATTFailedEvent_t {
    DeviceHistory_t deviceHistory;
    CallingDeviceID_t callingDevice;
} ATTFailedEvent_t;
```

Requesting private data

To request a specific version, or versions, of private data, an application allocates buffer space for working with private data, and it must pass negotiation information in the private data parameter of **acsOpenStream( )**. Here are a few tips for reading the Sample code for requesting private data on page 170.

- To indicate that the private data is to negotiate the version, the application sets the vendor field in the Private Data structure to the null-terminated string **“VERSION”**.
- The application specifies the acceptable vendor(s) and version(s) in the data field of the private data. The data field contains a one byte manifest constant **PRIVATE_DATA_ENCODING** followed by a null-terminated ASCII string containing a list of vendors and versions.
- When opening a TSAPI version 2 stream, an application should provide a list of supported private data versions in the data portion of the private data buffer. The AE Services TSAPI SDK provides the **attMakeVersionString()** function to simplify formatting this list. The sample code illustrates how to format the private data buffer to request private data version 3 through 8

Sample code for requesting private data

```
/* Define variable to hold Stream handle */
  ACSHandle_t  acsHandle ;

  /* Define private data buffer to hold Open Stream version request */
  ATTPrivateData_t privateData;

  /* Prepare the private data buffer for version request */
  strcpy( privateData.vendor, "VERSION" );
  privateData.data[0]= PRIVATE_DATA_ENCODING;

  /*
   * Encode private data version request.
   * Parameters specify that any of the private data versions in the range
   * 3 through 8 are acceptable to this client.
   * Note that the Open Stream acknowledgement indicates specifically which version
   * was negotiated and assigned to this stream.
   */
  attMakeVersionString( "3-8", &(privateData.data[1]) );
  privateData.length= strlen( &(privateData.data[1]+2) );

  /* Ask to open a TSAPI Service stream */
  RetCode_t rc= acsOpenStream(
    &acsHandle,
    APP_GEN_ID,  // application wants control over setting invoke IDs
    (Invoke_id_t)0, // (arbitrary) app sends '0' as the Invoke ID
    ST_CSTA,  // required parameter
    &advertisingName, // TLINK name like "AVAYA#SWITCH1#CSTA#SERVERNAME1"
    &loginID, &passwd, // authentication login and password
    ACS_LEVEL1,  // required parameter
    &version,  // required parameter set to "TS2"
    (WORD)0,  // send queue size
    (WORD)5,  // send extra bufs
    (WORD)50,  // receive queue size
    (WORD)5,  // receive extra bufs
    (ATTPrivateData_t*) &privateData
       // buffer of private data
   );
```

Applications that do not use private data

An application that does not use Private Data should not pass any private data to the **acsOpenStream( )** request.

The TSAPI Service interprets the lack of private data in the open stream request to mean that the application does not want private data. The TSAPI Service will then refrain from sending private data on that stream. This will saves LAN bandwidth that the private data would otherwise consume.

CSTA Get API Capabilities confirmation structures for Private Data Version 8

The TSAPI Service provides information about version-dependent private services and events in the CSTAGetAPICaps Confirmation private data interface. For Private Data Version 8 the ATTGetAPICapsConfirmation Event has been updated to include the singleStepTransfer field.

Field	Description
`unsigned char singleStepTransfer;` **NOTE:** This field was previously named `reserved1` .	Indicates whether the single step transfer call feature is available.

Code for the ATTGetAPICapsConfEvent - PDV 8

The ATT_Private_Identifiers.h file, which is provided in the AE Services TSAPI SDK contains the code for ATTGetAPICapsConfEvent. Here is the code for the ATTGetAPICapsConfEvent.

```
typedef struct ATTGetAPICapsConfEvent_t {
    char            switchVersion[65];
    unsigned char   sendDTMFTone;
    unsigned char   enteredDigitsEvent;
    unsigned char   queryDeviceName;
    unsigned char   queryAgentMeas;
    unsigned char   querySplitSkillMeas;
    unsigned char   queryTrunkGroupMeas;
    unsigned char   queryVdnMeas;
    unsigned char   singleStepConference;
    unsigned char   selectiveListeningHold;
    unsigned char   selectiveListeningRetrieve;
    unsigned char   setBillingRate;
    unsigned char   queryUCID;
    unsigned char   chargeAdviceEvent;
    unsigned char   singleStepTransfer;
    unsigned char   reserved2;
    unsigned char   deviceHistoryCount;
    char            adminSoftwareVersion[256];
    char            softwareVersion[256];
    char            offerType[256];
    char            serverType[256];
} ATTGetAPICapsConfEvent_t;
```

Private Data Service sample code

To retrieve private data return parameters from Communication Manager, the application must specify a pointer to a private data buffer as a parameter to either the **acsGetEventBlock()** or **acsGetEventPoll()** request.

When Communication Manager returns the private data, the application passes the address to **attPrivateData()** for decoding.

The following coding examples depict how these operations are carried out.

- Sample Code - Make Direct Agent Call on page 173
- Sample Code - Make Direct Agent Call (continued) on page 174
- Sample Code - Query ACD Split on page 177

Sample Code - Make Direct Agent Call

```
#include <stdio.h>

#include <acs.h>
#include <csta.h>
#include <attpriv.h>

/*
* Make Direct Agent Call  - from "12345" to  ACD Agent extension "11111"
*           - ACD agent must be logged into split "22222"
*           - no User to User info
*           - not a priority call
*/

ACSHandle_t acsHandle;              // An opened ACS Stream Handle
InvokeID_t  invokeID = 1;           // Application generated
                                    // Invoke ID
DeviceID_t  calling = "12345";      // Call originator, an on-PBX
                                    // extension
DeviceID_t  called  = "11111";      // Call destination, an ACD
                                    // Agent extension
DeviceID_t  split = "22222";        // ACD Agent is logged into
                                    // this split
Boolean     priorityCall = FALSE;   // Not a priority call
RetCode_t   retcode;                // Return code for service
                                    // requests
CSTAEvent_t cstaEvent;              // CSTA event buffer
unsigned    short       eventBufSize; // CSTA event buffer size
ATTPrivateData_tprivateData;        // ATT service request private
                                    // data buffer

            retcode = attDirectAgentCall(&privateData, &split, priorityCall,
                        NULL);

            if ( retcode < 0 ) {
            /* Some kind of failure, need to handle this ... */
            }
            retcode = cstaMakeCall(acsHandle, invokeID, &calling, &called,
```

Sample Code - Make Direct Agent Call (continued)

```
(PrivateData_t *)&privateData);

            if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}
/*  Make Call request succeeded.  Wait for confirmation event. */

eventBufSize = sizeof(CSTAEvent_t);
privateData.length = ATT_MAX_PRIVATE_DATA;

retcode = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
            &eventBufSize, (PrivateData_t *)&privateData, NULL);

if (retcode != ACSPOSITIVE_ACK) {
/* Some kind of failure, need to handle this ... */
}

if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
            (cstaEvent.eventHeader.eventType == CSTA_MAKE_CALL_CONF)) {
              if (cstaEvent.event.cstaConfirmation.invokeID == 1) {
            /*
             * Invoke ID matches, Make Call is confirmed.
            /*
                    }
            }
```

Sample Code - Set Agent State

```
#include <stdio.h>

#include <acs.h>
#include <csta.h>
#include <attpriv.h>

/*
 *  Set Agent State - Request to log in ACD Agent with initial work mode
 *  "Auto-In".
 */

ACSHandle_t acsHandle;              // An opened ACS Stream Handle
InvokeID_t  invokeID = 1;           // Application generated
                                    // Invoke ID
DeviceID_t  device = "12345";       // Device associated with
                                    // ACD Agent
AgentMode_t agentMode = AM_LOG_IN;  // Requested Agent Mode
AgentID_t   agentID = "01";         // Agent login identifier
AgentGroup_tagentGroup = "11111";   // ACD split to log Agent into
AgentPassword_t *agentPassword = NULL;// No password, i.e., not EAS
RetCode_t   retcode;                // Return Code for service
                                    // requests
CSTAEvent_t cstaEvent;              // CSTA event buffer
unsigned shorteventBufSize;         // CSTA event buffer size
ATTPrivateData_tprivateData;        // ATT service request private
// data buffer

retcode = attV6SetAgentState(&privateData, WM_AUTO_IN, 0, TRUE);

if (retcode < 0 ) {
            /* Some kind of failure, need to handle this ... */
            }

retcode = cstaSetAgentState(acsHandle, invokeID, &device, agentMode,
            &agentID, &agentGroup, agentPassword,
            (PrivateData_t *)&privateData);

if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}

}
```

Sample Code - Set Agent State (Continued)

```
/* Set Agent State request succeeded.  Wait for confirmation event.*/

eventBufSize = sizeof(CSTAEvent_t);
privateData.length = ATT_MAX_PRIVATE_DATA;

retcode = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
            &eventBufSize, (PrivateData_t *)&privateData, NULL);

if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}

if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
            (cstaEvent.eventHeader.eventType == CSTA_SET_AGENT_STATE_CONF)) {
             if (cstaEvent.event.cstaConfirmation.invokeID == 1) {
            /*
            * Invoke ID matches, Set Agent State is confirmed.
            * Private data is returned in confirmation event.
            */
            if (privateData.length > 0) {
            /* Confirmation contains private data */

            if (attPrivateData(&privateData, &attEvent) != ACSPOSITIVE_ACK) {
            /* Decoding error */
            }
            else { // See whether the requested change is pending or not
            ATTSetAgentStateConfEvent_t *setAgentStateConf ;
            SetAgentStateConf = &privateData.u.setAgentState;
            if (SetAgentStateConf->isPending == TRUE)
            // The request is pending
}
                    }
                        }
```

Sample Code - Query ACD Split

```
#include <stdio.h>

#include <acs.h>
#include <csta.h>
#include <attpriv.h>

/*
  * Query ACD Split via cstaEscapeService()
 */

ACSHandle_t acsHandle;              // An opened ACS Stream Handle
InvokeID_t  invokeID = 1;           // Application generated
                                    // Invoke ID
DeviceID_t  deviceID = "12345";     // Device associated with
                                    // ACD split
RetCode_t   retcode;                // Return code for service
                                    // requests
CSTAEvent_t cstaEvent;              // CSTA event buffer
unsigned shorteventBufSize;         // CSTA event buffer size
ATTPrivateData_tprivatedata;        // ATT private data buffer
ATTEvent_t  attEvent;               // ATT event buffer
ATTQueryAcdSplitConfEvent_t         // Query ACD Split confirmation
            *queryAcdSplitConf;     // event pointer


retcode = attQueryAcdSplit(&privatedata, &deviceID);

if (retcode < 0 ) {
            /* Some kind of failure, need to handle this ... */
}

retcode = cstaEscapeService(acsHandle, invokeID,
            (PrivateData_t *)&privateData);

if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}
```

Sample Code - Query ACD Split (Continued)

```
/*
* Now wait for confirmation event.
*
* To retrieve private data return parameters for Query ACD Split,
* the application must specify a pointer to a private data buffer as
* a parameter to either the acsGetEventBlock() or acsGetEventPoll()
* request. Upon return, the application passes the address
* to attPrivateData() for decoding.
*/

eventBufSize = sizeof(CSTAEvent_t);
privateData.length = ATT_MAX_PRIVATE_DATA;

retcode = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
            &eventBufSize, (PrivateData_t *)&privateData, NULL);

if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}

if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
            (cstaEvent.eventHeader.eventType == CSTA_ESCAPE_SVC_CONF)) {
              if (cstaEvent.event.cstaConfirmation.invokeID != 1) {
            /* Error - wrong invoke ID */
            }
            else if (privateData.length > 0) {
            /* Confirmation contains private data */

if (attPrivateData(&privateData, &attEvent) != ACSPOSITIVE_ACK) {
            /* Decoding error */
            }
            else if (attEvent.eventType == ATT_QUERY_ACD_SPLIT_CONF) {
            queryAcdSplitConf = (ATTQueryAcdSplitConfEvent_t *)
            &attEvent.u.queryAcdSplit;
            }
            }
            else {
            /* Error - no private data in confirmation event */
            }
}
```

Upgrading and maintaining applications that use private data

Private data version control refers to the method the TSAPI Service uses for maintaining multiple versions of private data. Private data version control provides you with a means of selecting the version of private data that is compatible with your application. If your applications use private data, be sure to read the following sections.

Note:

The AE Services 4.1 TSAPI Service is at PDV 8. Any TSAPI applications developed with the AE Services Release 4.1 TSAPI service should be written against PDV 8.

Using the private data header files

The private data header files (attpdefs.h and attpriv.h) are two important tools for using private data.

The attpdefs.h file -- PDU names and numbers

The **attpdefs.h** file contains the definitions of the Protocol Data Units (PDUs) that are used for private data version control. Each PDU in the attpdefs.h file has a PDU number associated with it. The PDU numbers with the highest values represent the latest version of private data for a given service, confirmation event, or unsolicited event. Here are a few examples of #define statements in the attpdefs.h file to illustrate this point.

#define ATT_SINGLE_STEP_TRANSFER_CALL 142

#define ATT_FAILED 141

#define ATTV7_FAILED 137

#define ATT_QUERY_DEVICE_NAME_CONF 125 - the highest private data version of the Query device name confirmation event, which is Private Data Version 7 (this event did not change for PDV 8).

#define ATTV6_QUERY_DEVICE_NAME_CONF 89 - the previous private data version of the Query device name service, PDV 6.

#define ATT_ROUTE_SELECT 126 - the highest private data version of the Route Select service, which is Private Data Version 7 (this service did not change for PDV 8).

#define ATTV6_ROUTE_SELECT 116 - the previous private data version of the Route Select service, PDV 6)

The attpriv.h file -- other related PDU elements

The **attpriv.h** file contains the PDU structures for the PDUs that are defined in the attpdefs.h file. Table 12 contains examples from both header files. Here are a few fundamental points to notice about the elements in the private data header files:

- PDU names without version qualifiers (ATT_QUERY_DEVICE_NAME_CONF) represent the highest version of private data. PDU names with version qualifiers (ATTV6_QUERY_DEVICE_NAME_CONF) indicate a specific version of private data.
- PDU structure names without version qualifiers (ATTQueryDeviceNameConfEvent_t) represent the highest version of private data. PDU structure names with version qualifiers (ATTV6QueryDeviceNameConfEvent_t) indicate a specific version of private data.
- PDU union member names without version qualifiers (queryDeviceName) represent the highest version of private data. PDU union member names with version qualifiers (v6queryDeviceName) indicate a specific version of private data.
- Function names behave differently.
  - Function names for service requests use a version qualifier to denote the highest version of private data for that particular service. For example, Route Select **attV7RouteSelect()**, Make Call (**attV6MakeCall()**). The only time a function name is unqualified is when it is initially introduced. When you request the latest private data version you always get the highest version of a service request.

Table 12: Elements in private data header files

PDU name and number attpdefs.h	related elements in attpriv.h
ATT_QUERY_DEVICE_NAME_CONF 125	• **ATTQueryDeviceNameConfEvent_t** (structure name) • **queryDeviceName** (union member name)
ATTV6_QUERY_DEVICE_NAME_CONF 89	• **ATTV6QueryDeviceNameConfEvent_t** (structure name) • **v6queryDeviceName** (union member name)
ATT_ROUTE_SELECT 126	• **ATTRouteSelect_t** (structure name) • **routeSelectReq** (union member name) • **attV7RouteSelect()** (function name)
ATTV6_ROUTE_SELECT 116	• **ATTV6RouteSelect_t** (structure name) • **v6routeSelectReq** (union member name) • **attV6RouteSelect()**(function name)

Upgrading PDV 6 applications to PDV 7

If you have an existing application that was developed to PDV 6, and you want to use PDV 7 functionality, you will need to need to upgrade your application to take advantage of the PDV 7 features, and then recompile your application with the PDV 7 SDK. The following steps outline the high level tasks necessary for upgrading a PDV 6 application to PDV 7.

1. Make sure you have installed the AE Services 3.1 or AE Services 4.1 TSAPI SDK. Whenever you recompile your application, or applications, you must use the AE Services 3.1 or AE Services 4.1 TSAPI SDK, which is at PDV 7.
2. Use Table 24: Private Data Version 7 features on page 814 to help you determine what PDV 7 functionality you want to incorporate into your application.
3. Make the coding level changes in your application, as follows:
   - Wherever your code includes references to the private data ***function name*** for the **Route Select** service, you must change it to **attV7RouteSelect()**.
4. Change **acsOpenStream** to negotiate PDV 7. See Requesting private data on page 169.
5. Recompile your application with AE Services 3.1 or AE Services 4.1 TSAPI SDK, which is at PDV 7.

Things you do not need to change in your code

If you are upgrading your application to PDV 7, you do not need to change the following.

- Private data PDU names, and their corresponding PDU structure names, and union member names for confirmations and unsolicited events (for example, Query Device Name, Query Station Status, and Query Trunk Group).

Maintaining applications that use prior versions of private data

Programming environments that support a mix of applications often include applications that are written to different private data versions. Although the recommended practice is to upgrade your applications to the latest private data version, there might be cases where you need to maintain older applications.

Maintaining a PDV 7 application in a PDV 8 environment

To maintain a PDV 7 application in an AE Services 4.1 PDV 8 environment you will need to make coding level changes to your application, and then recompile your application with the PDV 8 library. The following steps outline the high level tasks necessary for maintaining a PDV 7 application in a PDV 8 environment.

1. Make sure you have installed the AE Services 4.1 TSAPI SDK.
2. Make the coding level changes in your application.
   - Change any private data PDU names, along with their corresponding PDU structure names, and union member names in your application as indicated in the following table.

If your code contains these PDUs and structure member names	Rename them as follows:
ATT_FAILED ATTFailedEvent_t failedEvent	ATTV7_FAILED ATTV7FailedEvent_t v7failedEvent

3. Recompile your application with AE Services 4.1 TSAPI SDK.

Things you do not need to change in your code

If you are maintaining a PDV 7 application in a PDV 8 environment, you do not need to change the following.

- The open stream request. Your applications will continue to negotiate a PDV 7 stream.

Recompiling against the same SDK

If you have an existing application that was developed with an earlier version of the SDK, and you do not foresee making use of capabilities available in newer private data versions, then you may simply continue to compile your application with the earlier version of the SDK.

For example if you need to change your program for a bug fix, and you are not changing any private data related code, you would recompile it with the original SDK. If you developed the application with the PDV 6 SDK, you would recompile with the PDV 6 SDK, as opposed to a PDV 7 SDK. As long as you use this method to recompile, you do not have to make any private data related coding changes.

Chapter 6: Call Control Service Group

Call Control Service Group describes the services that enable a telephony client application to control a call or connection on Communication Manager. These services are typically used for placing calls from a device and controlling any connection on a single call as the call moves through Communication Manager.

Tip:
Although client applications can manipulate switch objects, Call Control Services do not provide Event Reports as objects change state. To monitor switch object state changes (that is, to receive CSTA Event Report Services from a switch), a client must request a CSTA Monitor Service for an object before it requests Call Control Services for that object.

This chapter includes the following topics.

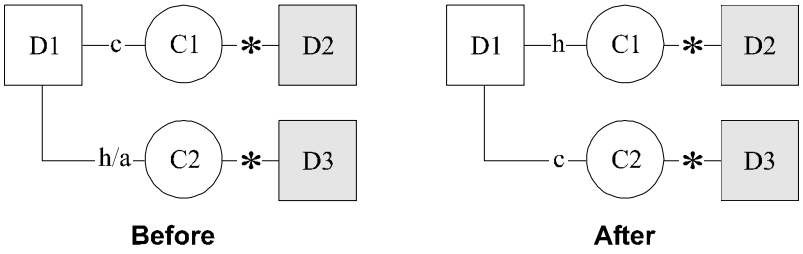
Graphical Notation Used in the Diagrams

The diagrams in this chapter use the following graphical notation.

- Boxes represent devices and D1, D2, and D3 represent deviceIDs.
- Circles represent calls and C1, C2, and C3 represent callIDs.
- Lines represent connections between a call and a device; and C1-D1, C1-D2, C2- D3, etc., represent connectionIDs.
- The absence of a line is equivalent to a connection in the Null connection state.
- Labels in boxes and circles represent call and device instances.
- Labels on lines represent a connection state using the following key:
  - a = Alerting
  - c = Connected
  - f = Failed
  - h = Held
  - i = Initiated
  - q = Queued
  - a/h = Alerting or Held
  - * = Unspecified
- Grayed boxes represent devices in a call unaffected by the service or event report.
- White boxes and circles represent devices and calls affected by the service or event report.
- The parameters for the function call of the service are indicated in bold italic font.

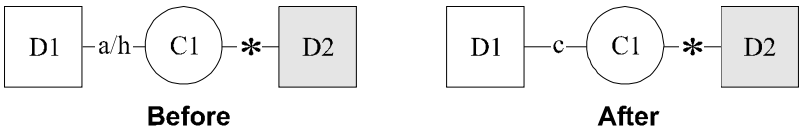
Alternate Call Service

The Alternate Call Service provides a compound action of the Hold Call Service followed by Retrieve Call Service/Answer Call. The Alternate Call Service places an existing activeCall (C1-D1) at a device to another device (D2) on hold and, in a combined action, retrieves/establishes a held/delivered otherCall (C2-D1) between the same device D1 and another device (D3) as the active call. Device D2 can be considered as being automatically placed on hold immediately prior to the retrieval/establishment of the held/alerting call to device D3. A successful service request will cause the held/alerting call to be swapped with the active call.



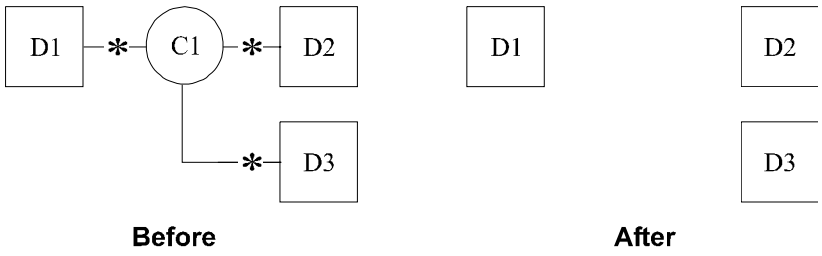
Answer Call Service

The Answer Call Service is used to answer an incoming call (C1) that is alerting a device (D1) with the connection alertingCall (C1-D1). This service is typically used with telephones that have attached speakerphone units to establish the call in a hands-free operation. The Answer Call Service can also be used to retrieve a call (C1) that is held by a device (D1) with the connection alertingCall (C1-D1).



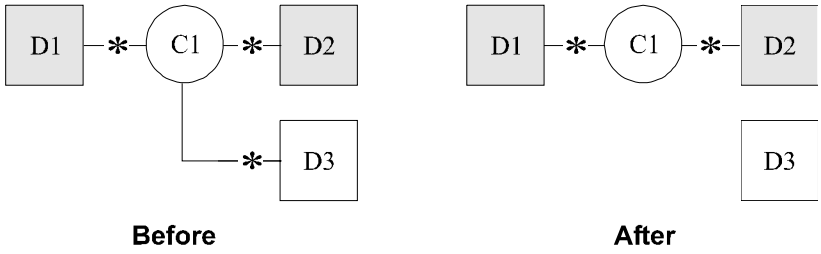
Clear Call Service

This service will cause each device associated with a call (C1) to be released and the connectionIDs (and their components) to be freed.



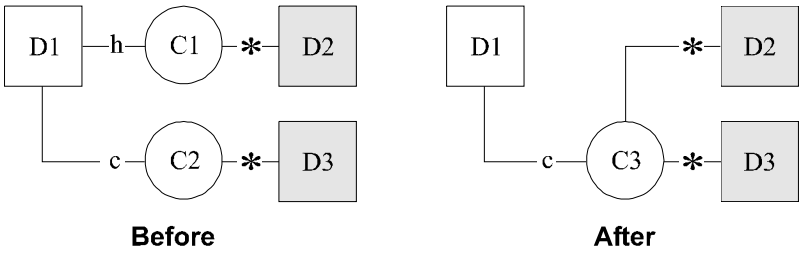
Clear Connection Service

This service releases the specified connection, call (C1-D3), and its connectionID instance from the designated call (C1). The result is as if the device had hung up on the call. The phone does not have to be physically returned to the switchhook, which may result in silence, dial tone, or some other condition. Generally, if only two connections are in the call, the effect of cstaClearConnection is the same as cstaClearCall. Note that it is likely that the call (C1) is not cleared by this service if it is some type of conference.



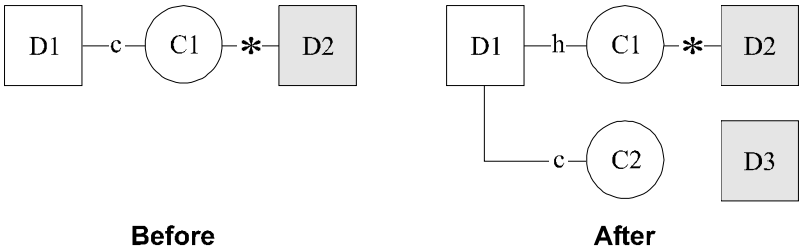
Conference Call Service

This service provides the conference of an existing heldCall (C1-D1) and another activeCall (C2-D1) at the same device. The two calls are merged into a single call (C3) and the two connections (C1-D1, C2-D1) at the conferencing device (D1) are resolved into a single connection, newCall (C3-D1), in the Connected state.



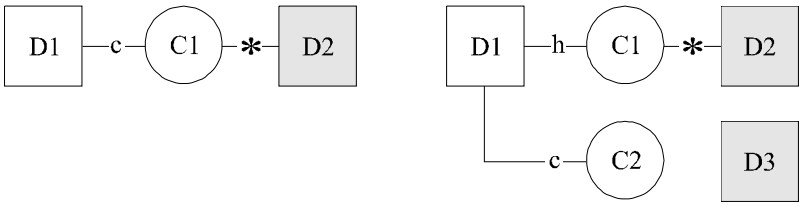
Consultation Call Service

The Consultation Call Service will provide the compound action of the Hold Call Service followed by Make Call Service. This service places an active activeCall (C1-D1) at a device (D1) on hold and initiates a new call from the same device D1 to another calledDevice (D3). The client is returned with the connection newCall (C2-D1).



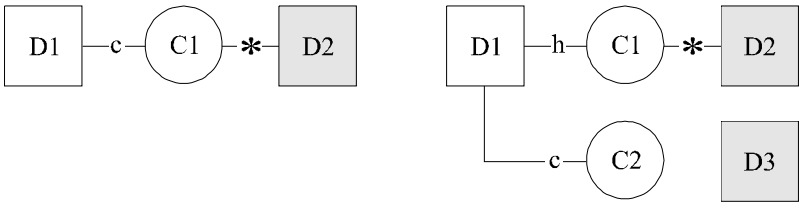
Consultation Direct-Agent Call Service

The Consultation Direct-Agent Call Service will provide the compound action of the Hold Call Service followed by Make Direct-Agent Call Service. This service places an active activeCall (C1-D1) at a device (D1) on hold and initiates a new direct-agent call from the same device D1 to another calledDevice (D3). The client is returned with the connection newCall (C2-D1).



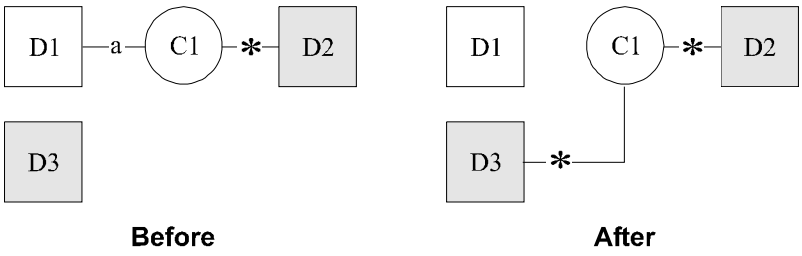
Consultation Supervisor-Assist Call Service

The Consultation Supervisor-Assist Call Service will provide the compound action of the Hold Call Service followed by Make Supervisor-Assist Call Service. This service places an active activeCall (C1-D1) at a device (D1) on hold and initiates a new supervisor-assist call from the same device D1 to another calledDevice (D3). The client is returned with the connection newCall (C2-D1).



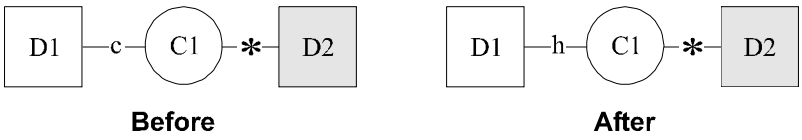
Deflect Call Service

The Deflect Call Service redirects an alerting call (C1) at a device (D1) with the connection deflectCall to a new destination, either on-PBX or off-PBX.



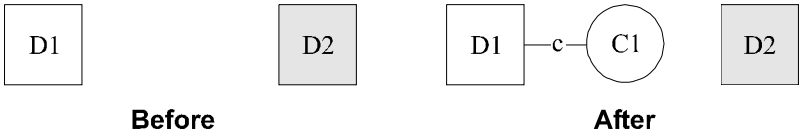
Hold Call Service

The Hold Call Service places a call (C1) at a device (D1) with the connection activeCall (C1-D1) on hold. The effect is as if the specified party depressed the hold button on the device or flashed the switchhook to locally place the call on hold. The call is usually in the active state. This service maintains a relationship between the holding device (D1) and the held call (C1) that lasts until the call is retrieved from the hold status or until the call is cleared.



Make Call Service

The Make Call Service originates a call between two devices designated by the application. When the service is initiated, the callingDevice (D1) is prompted (if necessary), and when that device acknowledges, a call to the calledDevice (D2) is originated. A call is established as if D1 had called D2, and the client is returned with the connection newCall (C1-D1).



Make Direct-Agent Call Service

The Make Direct-Agent Call Service originates a call between two devices: a user station and an ACD agent logged into a specified split. When the service is initiated, the callingDevice (D1) is prompted (if necessary), and when that device acknowledges, a call to the calledDevice (D2) is originated. A call is established as if D1 had called D2, and the client is returned with the connection newCall (C1-D1).

Before **After**

The Make Direct Agent Call Service should be used only in the following two situations:

- Direct Agent Calls in a non-EAS environment
- Direct Agent Calls in an EAS environment only when it is required to ensure that these calls against a skill other than that skill specified for these measurements on the DEFINITY PBX for that agent.

Preferably in an EAS environment, Direct Agent Calls can be made using the Make Call service and specifying an Agent login-ID as the destination device. In this case Direct Agent Calls will be measured against the skill specified or those measurements on the DEFINITY PBX for that agent.

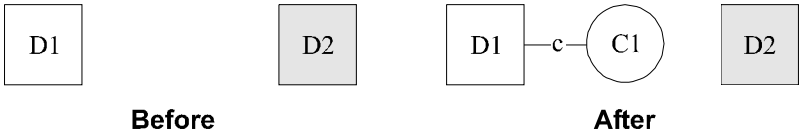
Make Predictive Call Service

The Make Predictive Call Service originates a Switch-Classified call between two devices. The service attempts to create a new call and establish a connection with the calledDevice (D2) first. The client is returned with the connection newCall (C1-D2).

Before **After**

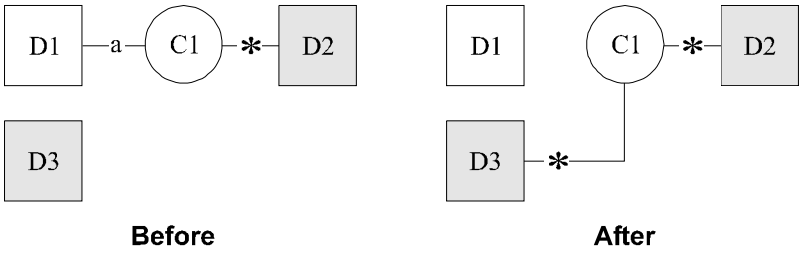
Make Supervisor-Assist Call Service

The Make Supervisor-Assist Call Service originates a supervisor-assist call between two devices: an ACD agent station and another station (typically a supervisor). When the service is initiated, the callingDevice (D1) is prompted (if necessary), and when that device acknowledges, a call to the calledDevice (D2) is originated. A call is established as if D1 had called D2, and the client is returned with the connection newCall (C1-D1).



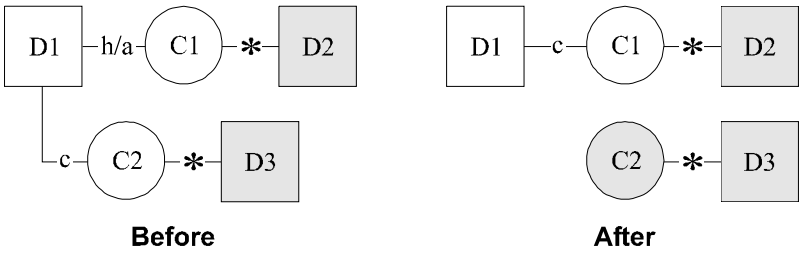
Pickup Call Service

The Pickup Call Service takes an alerting call (C1) at a device (D1) with the connection deflectCall to another on-PBX device.



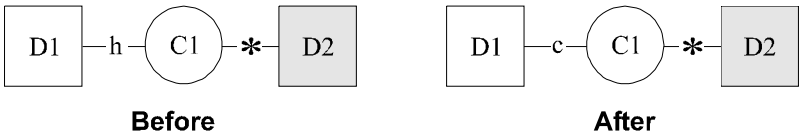
Reconnect Call Service

The Reconnect Call Service allows a client to disconnect an existing connection activeCall (C2-D1) from a call and then retrieve/establish a previously held/delivered connection heldCall (C1-D1).



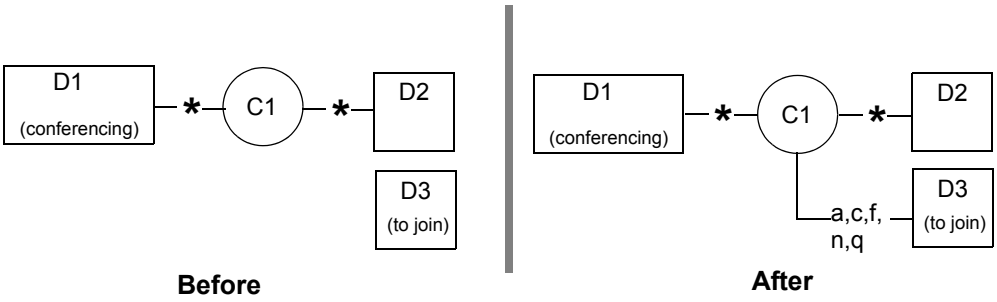
Retrieve Call Service

The service restores a held connection heldCall (C1-D1) to the Connected state (active).



Single Step Conference

The single step conference collapses the two steps of the conference call process into one.

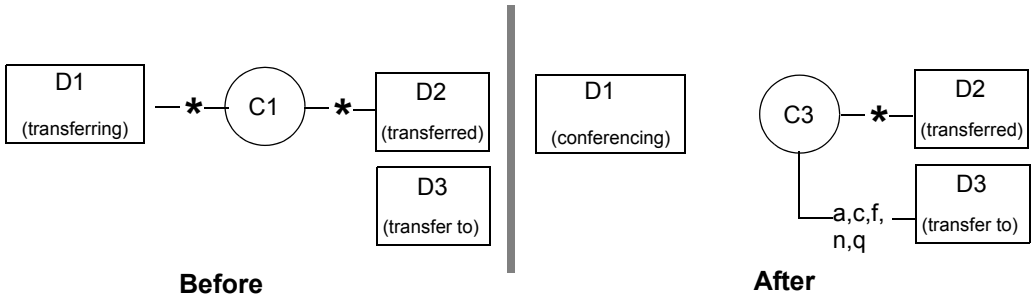


By specifying D3 as the destination for a single step conference involving call C1, the connection D3C1 is created exactly the same way that it would be if any of the devices already in C1 had just placed a new call to D3 using the Make Call service. The difference is that all of the devices already in C1 remain in the call.

Single Step Transfer Call

The Single Step Transfer Call service transfers an existing connection to another device. This transfer is performed in a single step. This means that the device transferring the call does not have to place the existing call on hold before issuing the Single Step Transfer Call service.

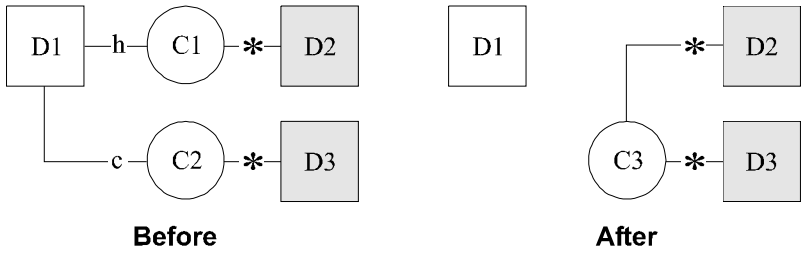
The connection being transferred may be in the Alerting, Connected, Held, or Queued state.



In a single step, this service drops D1 from the call it is transferring (C1), places a new call (C3) to the transferred-to device (D3) and merges the remaining devices from C1 into C3. When the service request is complete the result appears as if D2 had used the Make Call Service to call D3 directly. This state of connection D3C3 is the same as described for the called connection after successful completion of a make call service.

Transfer Call Service

This service provides the transfer of a heldCall (C1-D1) with an activeCall (C2-D1) at the same device (D1). The transfer service merges two calls (C1, C2) with connections (C3-D2, C3-D3) at a single common device (D1) into one call (C3). Also, both of the connections to the common device become Null and their connectionIDs are released. When the transfer completes, the common device (D1) is released from the calls (C1, C2). A callID, newCall (C3) that specifies the resulting new call for the transferred call is provided.



Alternate Call Service

Summary

- Direction: Client to Switch
- Function: *cstaAlternateCall ()*
- Confirmation Event: *CSTAAlternateCallConfEvent*
- Service Parameters: *activeCall, otherCall*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

The Alternate Call Service allows a client to put an existing active call (activeCall) on hold and then answer an alerting (or bridged) call or retrieve a previously held call (otherCall) at the same station. It provides the compound action of the Hold Call Service followed by an Answer Call Service or a Retrieve Call Service.

The Alternate Call Service request is acknowledged (Ack) by the switch if the switch is able to put the activeCall on hold and

- connect the specified alerting otherCall either by forcing the station off-hook (turning the speakerphone on) or waiting up to five seconds for the user to go off- hook, or
- retrieve the specified held otherCall.

The request is negatively acknowledged if the switch:

- fails to put activeCall on hold (for example, call is in alerting state),
- fails to connect the alerting otherCall (for example, call dropped), or
- fails to retrieve the held otherCall.

If the request is negatively acknowledged, the TSAPI Service will attempt to put the activeCall to its original state, if the original state is known by the TSAPI Service before the service request. If the original state is unknown, there is no recovery for the activeCall's original state.

Service Parameters

:

activeCall	[mandatory] A valid connection identifier that indicates the callID and the station extension (STATIC_ID). The deviceID in activeCall must contain the station extension of the controlling device. The local connection state of the call can be either active or held.
otherCall	[mandatory] A valid connection identifier that indicates the callID and the station extension (STATIC_ID). The deviceID in otherCall must contain the station extension of the controlling device. The local connection state of the call can be either alerting, bridged, or held.

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 • INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in activeCall or otherCall. • INVALID_CSTA_CONNECTION_IDENTIFIER (13) An incorrect callID, a incorrect deviceID, or dynamic device ID type is specified in activeCall or otherCall. • GENERIC_STATE_INCOMPATIBILITY (21) The otherCall station user did not go off-hook within five seconds and is not capable of being forced off-hook. • INVALID_OBJECT_STATE (22) The otherCall is not in the alerting, connected, held, or bridged state. • INVALID_CONNECTION_ID_FOR_ACTIVE_CALL (23) The controlling deviceID in activeCall and otherCall is different. • NO_ACTIVE_CALL (24) The activeCall to be put on hold is not currently active (in alerting state, for example) so it cannot be put on hold. • NO_CALL_TO_ANSWER (28) The otherCall was redirected to coverage within the five- second interval.

- GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31) The client attempted to add a seventh party (otherCall) to a call with six active parties.
- RESOURCE_BUSY (33) User at the otherCall station is busy on a call or there is no idle appearance available. It is also possible that the switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, etc.) to the same device.
- OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44) The client attempted to put a third party (activeCall) on hold (two parties are on hold already) on an analog station.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in activeCall or otherCall.

Detailed Information:

See Detailed Information: in the “Answer Call Service” section and Detailed Information: in the “Hold Call Service” section in this chapter.

Syntax

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

// cstaAlternateCall() - Service Request

RetCode_t       cstaAlternateCall (
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    ConnectionID_t  *activeCall,    // devIDType= STATIC_ID
    ConnectionID_t  *otherCall,     // devIDType= STATIC_ID
    PrivateData_t   *privateData);

// CSTAAlternateCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass; // CSTACONFIRMATION
    EventType_t     eventType;  // CSTA_ALTERNATE_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAAlternateCallConfEvent_t    alternateCall;
            } u;

        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAAlternateCallConfEvent_t {
    Nulltype        null;
} CSTAAlternateCallConfEvent_t;
```

Answer Call Service

Summary

- Direction: Client to Switch
- Function: *cstaAnswerCall ()*
- Confirmation Event: *CSTAAnswerCallConfEvent*
- Service Parameters: *alertingCall*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description

The Answer Call Service allows a client application to request on behalf of a station user the ability to answer a ringing or bridged call (alertingCall) present at a station. Answering a ringing or bridged call means to connect a call by forcing the station off-hook if the user is on-hook, or cutting the call through to the head or handset if the user is off-hook (listening to dial tone or being in the off-hook idle state). The effect is as if the station user selected the call appearance of the alerting or bridged call and went off-hook.

The deviceID in alertingCall must contain the station extension of the endpoint to be answered on the call. A Delivered Event Report must have been received by the application prior to this request.

The Answer Call Service can be used to answer a call present at any station type (for example, analog, DCP, hybrid, and BRI).

The Answer Call Service request is acknowledged (Ack) by the switch if the switch is able to connect the specified call either by forcing the station off-hook (turning on the speakerphone) or waiting up to five seconds for the user to go off-hook. Answering a call that is already connected or in the held state will result in a positive acknowledgment and, if the call was held, the call becomes connected.

Service Parameters:

alertingCall	[mandatory] A valid connection identifier that indicates the callID and the station extension (STATIC_ID).

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in alertingCall.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) An incorrect callID or an incorrect deviceID is specified.
- GENERIC_STATE_INCOMPATIBILITY (21) The station user did not go off-hook within five seconds and is not capable of being forced off-hook.
- INVALID_OBJECT_STATE (22) The specified connection at the station is not in the alerting, connected, held, or bridged state.
- NO_CALL_TO_ANSWER (28) The call was redirected to coverage within the five-second interval.
- GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31) The client attempted to add a seventh party to a call with six active parties.
- RESOURCE_BUSY (33) The user at the station is busy on a call or there is no idle appearance available.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in alertingCall.

Detailed Information:

- Multifunction Station Operation - For a multifunction station user, this service will be successful in the following situations:
  - The user's state is being alerted on-hook. For example, the user can either be forced off-hook or is manually taken off-hook within five seconds of the request. The switch will select the ringing call appearance.
  - The user is off-hook idle. The switch will select the alerting call appearance and answer the call.
  - The user is off-hook listening to dial tone. The switch will drop the dial tone call appearance and answer the alerting call on the alerting call appearance.

A held call will be answered (retrieved) on the held call appearance, provided that the user is not busy on another call. This service is not recommended to retrieve a held call. The cstaRetrieveCall Service should be used instead.

A bridged call will be answered on the bridged call appearance, provided that the user is not busy on another call, or the exclusion feature is not active for the call.

An ACB, PCOL, or TEG call will be answered on a free call appearance, provided that the user is not busy on another call.

If the station is active on a call (talking), listening to reorder/intercept tone, or does not have an idle call appearance (for ACB, ICOM, PCOL, or TEG calls) at the time the switch receives the Answer Call Service request, the request will be denied.

- Analog Station Operation - For an analog station user, the service will be successful only under the following circumstances:
  - The user is being alerted on-hook (and is manually taken off-hook within five seconds).
  - The user is off-hook idle (or listening to dial tone) with a call waiting. The switch will drop the dial tone (if any) and answer the call waiting call.
  - The user is off-hook idle (or listening to dial tone) with a held call (soft or hard). The switch will drop the dial tone (if any) and answer the specified held call (there could be two held calls at the set, one soft-held and one hard-held).

An analog station may only have one or two held calls when invoking the Answer Call Service on a call. If there are two held calls, one is soft-held, the other hard-held. Answer Call Service on any held call (in the absence of another held call and with an off- hook station) will reset the switch-hook flash counter to zero, as if the user had manually gone on-hook and answered the alerting/held call. Answer Call Service on a hard-held call (in the presence of another, soft-held call and with an off-hook station) will leave the switch-hook flash counter unchanged. Thus, the user may use subsequent switch-hook flashes to effect a conference operation between the previously soft-held call and the active call (reconnected from the hard-held call). Answer Call Service on a hard-held call in the presence of another soft-held call and with the station on-hook will be denied. This is consistent with manual operation because when the user goes on-hook with two held calls, one soft-held and one hard-held, the user is again alerted, goes off-hook, and the soft-held call is retrieved.

If the station is active on a call (talking) or listening to reorder/intercept tone at the time the Answer Call Service is requested, the request will be denied (RESOURCE_BUSY).

Syntax

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

// cstaAnswerCall() - Service Request

RetCode_t       cstaAnswerCall (
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    ConnectionID_t  *alertingCall,  // devIDType= STATIC_ID
    PrivateData_t   *privateData);

// CSTAAnswerCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass; // CSTACONFIRMATION
    EventType_t     eventType;  // CSTA_ANSWER_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAAnswerCallConfEvent_t       answerCall;
            } u;

        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAAnswerCallConfEvent_t {
    Nulltype        null;
} CSTAAnswerCallConfEvent_t;
```

Clear Call Service

Summary

- Direction: Client to Switch
- Function: *cstaClearCall ()*
- Confirmation Event: *CSTAClearCallConfEvent*
- Service Parameters: *call*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

The Clear Call Service disconnects all connections from the specified call and terminates the call itself. All connection identifiers previously associated with the call are no longer valid.

Service Parameters:

call	[mandatory] A valid connection identifier that indicates the call to be cleared. The deviceID of call is optional. If it is specified, it is ignored.

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786. • NO_ACTIVE_CALL (24) The callID of the connectionID specified in the request is invalid.

Detailed Information:

- Switch operation - After a successful Clear Call Service request:
  - Every station dropped will be in the off-hook idle state.

- Any lamps associated with the call are off.
- Displays are cleared.
- Auto-answer analog stations do not receive dial tone.
- Manual-answer analog stations receive dial tone.

Syntax

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

// cstaClearCall() - Service Request

RetCode_t       cstaClearCall (
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    ConnectionID_t  *call,// deviceID, devIDType are ignored
    PrivateData_t   *privateData);

// CSTAClearCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass; // CSTACONFIRMATION
    EventType_t     eventType;  // CSTA_CLEAR_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAClearCallConfEvent_t        clearCall;
            } u;

        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAClearCallConfEvent_t {
    Nulltype        null;
} CSTAClearCallConfEvent_t;
```

Clear Connection Service

Summary

- Direction: Client to Switch
- Function: *cstaClearConnection()*
- Confirmation Event: *CSTAClearConnectionConfEvent*
- Private Data Function: *attV6ClearConnection()* (private data version 6), *attClearConnection()* (private data version 2, 3, 4, and 5)
- Service Parameters: *call*
- Private Parameters: *dropResource, userInfo*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description

The Clear Connection Service disconnects the specified device from the designated call. The connection is left in the Null state. The connection identifier is no longer associated with the call. The party to be dropped may be a station or a trunk.

A connection in the alerting state cannot be cleared.

Service Parameters

call	[mandatory] A valid connection identifier that indicates the endpoint to be disconnected.

Private Parameters:

dropResource [optional] Specifies the resource to be dropped from the call. The available resources are DR_CALL_CLASSIFIER and DR_TONE_GENERATOR. The tone generator is any Communication Manager applied denial tone that is timed by the switch.

userInfo [optional] Contains user-to-user information. This parameter allows an application to associate caller information, up to 32 or 96 bytes, with a call. This information may be a customer number, credit card number, alphanumeric digits, or a binary string.

It is propagated with the call when the call is dropped and passed to the application in a Connection Cleared Event Report. A NULL indicates this parameter is not present.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes.

Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE - There is no data provided in the data parameter.
- UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

Ack Parameter

:

noData None for this service.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786

- GENERIC_UNSPECIFIED (0) The specified data provided for the userInfo parameter exceeds the maximum allowable size. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. See the description of the userInfo parameter.
- INVALID_OBJECT_STATE (22) The specified connection at the station is not currently active (in alerting or held state) so it cannot be dropped.
- NO_ACTIVE_CALL (24) The connectionID contained in the request is invalid. CallID may be incorrect.
- NO_CONNECTION_TO_CLEAR (27) The connectionID contained in the request is invalid. CallID may be correct, but deviceID is wrong.
- RESOURCE_BUSY (33) The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, etc.) to the same device.

Detailed Information:

- Analog Stations - The auto-answer analog stations do not receive dial tone after a Clear Connection request. The manual answer analog stations receive dial tone after a Clear Connection request.
- Bridged Call Appearance - Clear Connection Service is not permitted on parties in the bridged state and may also be more restrictive if the principal of the bridge has an analog station or the exclusion option is in effect from a station associated with the bridge or PCOL.
- Drop Button Operation - The operation of this button is not changed with the Clear Connection Service.
- Switch Operation - When a party is dropped from an existing conference call with three or more parties (directly connected to the switch), the other parties remain on the call. Generally, if this was a two-party call, the entire call is dismantled. This is the case for typical voice processing. There is a Communication Manager feature "Return VDN Destination" where this is not true. In general, this feature will not be encountered in typical call processing

  Note:
  Only connected parties can be dropped from a call. Held, bridged, and alerting parties cannot be dropped by the Clear Connection Service.

- Temporary Bridged Appearance - The Clear Connection Service request is denied for a temporary bridged appearance that is not connected on the call.

Syntax

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

// cstaClearConnection() - Service Request

RetCode_t       cstaClearConnection (
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    ConnectionID_t  *call, // devIDType= STATIC_ID or
                                        // DYNAMIC_ID
    PrivateData_t   *privateData);

// CSTAClearConnectionConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_CLEAR_CONNECTION_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAClearConnectionConfEvent_t  clearConnection;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAClearConnectionConfEvent_t {
    Nulltype        null;
} CSTAClearConnectionConfEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6ClearConnection() - Service Request Private Data
                           // Setup Function

RetCode_t    attV6ClearConnection(
     ATTPrivateData_t      *privateData,
     ATTDropResource_t     dropResource);    // NULL indicates
                                             // no dropResource
                                             // specified
     ATTUserToUserInfo_t   *userInfo);       // NULL indicates
                                             // no userInfo
                                             // specified

typedef struct ATTPrivateData_t {
     char          vendor[32];
     ushort        length;
     char          data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTDropResource_t {
     DR_NONE               = -1,// indicates not specified
     DR_CALL_CLASSIFIER    = 0, // call classifier to be dropped
     DR_TONE_GENERATOR     = 1  // tone generator to be dropped }
ATTDropResource_t;

#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef struct    ATTUserToUserInfo_t  {
     ATTUUIProtocolType_t type;
     struct {
          short length;          // 0 indicates UUI not present
          unsigned char value[ATT_MAX_USER_INFO];
     } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
     UUI_NONE              = -1,// indicates not specified
     UUI_USER_SPECIFIC     = 0,  // user-specific
     UUI_IA5_ASCII         = 4   // null-terminated ascii
                                 // character string
} ATTUUIProtocolType_t;
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attClearConnection() - Service Request Private Data
                        // Setup Function

RetCode_t    attClearConnection(
    ATTPrivateData_t    *privateData,
    ATTDropResource_t   dropResource);  // NULL indicates
                                        // no dropResource
                                        // specified
    ATTUserToUserInfo_t *userInfo);     // NULL indicates
                                        // no userInfo
                                        // specified

typedef struct ATTPrivateData_t {
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTDropResource_t {
    DR_NONE             = -1,// indicates not specified
    DR_CALL_CLASSIFIER  = 0,// call classifier to be dropped
    DR_TONE_GENERATOR   = 1 // tone generator to be dropped }
ATTDropResource_t;

typedef structATTV5UserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
        short length;       // 0 indicates UUI not present
        unsigned char value[32];
    } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE            = -1,// indicates not specified
    UUI_USER_SPECIFIC   = 0, // user-specific
    UUI_IA5_ASCII       = 4  // null-terminated ascii
                             // character string
} ATTUUIProtocolType_t;
```

Conference Call Service

Summary

- Direction: Client to Switch
- Function: *cstaConferenceCall ()*
- Confirmation Event: *CSTAConferenceCallConfEvent*
- Private Data Confirmation Event: *ATTConferenceCallConfEvent* (private data version 5)
- Service Parameters: *heldCall, activeCall*
- Ack Parameters: *newCall, connList*
- Ack Private Parameters: *ucid*
- Nak Parameter: *universalFailure*

Functional Description

This service provides the conference of an existing held call (heldCall) and another active or proceeding call (alerting, queued, held, or connected) (activeCall) at a device provided that heldCall and activeCall are not both in the alerting state at the controlling device. The two calls are merged into a single call and the two connections at the conference-controlling device are resolved into a single connection in the connected state. The pre-existing CSTA connectionID associated with the device creating the conference is released, and a new callID for the resulting conferenced call is provided.

Service Parameters:

heldCall	[mandatory] Must be a valid connection identifier for the call that is on hold at the controlling device and is to be conferenced with the activeCall. The deviceID in heldCall must contain the station extension of the controlling device.
activeCall	[mandatory] Must be a valid connection identifier for the call that is active or proceeding at the controlling device and that is to be conferenced with the heldCall. The deviceID in activeCall must contain the station extension of the controlling device.

Ack Parameters:

newCall — [mandatory - partially supported] A connection identifier specifies the resulting new call identifier for the calls that were conferenced at the conference-controlling device. This connection identifier replaces the two previous call identifiers at that device.

connList — [optional - supported] Specifies the devices on the resulting newCall. This includes a count of the number of devices in the new call and a list of up to six connectionIDs and up to six deviceIDs that define each connection in the call.

- If a device is on-PBX, the extension is specified. The extension consists of station or group extensions. Group extensions are provided when the conference is to a group and the conference completes before the call is answered by one of the group members (TEG, PCOL, hunt group, or VDN extension). It may contain alerting extensions.
- The static deviceID of a queued endpoint is set to the split extension of the queue.
- If a party is off-PBX, then its static device identifier or its previously assigned trunk identifier is specified.

Ack Private Parameters

ucid — [optional] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure — If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in heldCall or activeCall.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) The controlling deviceID in heldCall or activeCall has not been specified correctly.
- GENERIC_STATE_INCOMPATIBILITY (21) Both calls are alerting or both calls are being service- observed or an active call is in a vector processing stage.
- INVALID_OBJECT_STATE (22) The connections specified in the request are not in the valid states for the operation to take place. For example, it does not have one call active and one call in the held state as required.

- INVALID_CONNECTION_ID_FOR_ACTIVE_CALL (23) The callID or deviceID in activeCall or heldCall has not been specified correctly.
- RESOURCE_BUSY (33) The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Conference Call, etc.) to the same device.
- CONFERENCE_MEMBER_LIMIT_EXCEEDED (38) The request attempted to add a seventh party to an existing six-party conference call. If a station places a six-party conference call on hold and another party adds yet another station (so that there are again six active parties on the call - the Communication Manager limit), then the station with the call on hold will not be able to retrieve the call.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in heldCall or activeCall.

Detailed Information:

- Analog Stations - Conference Call Service will only be allowed if one call is held and the second is active (talking). Calls on hard-hold or alerting cannot be affected by a Conference Call Service. An analog station will support Conference Call Service even if the “switch-hook flash” field on the Communication Manager system administered form is set to “no”. A “no” in this field disables the switch-hook flash function, meaning that a user cannot conference, hold, or transfer a call from his/her phone set, and cannot have the call waiting feature administered on the phone set.
- Bridged Call Appearance - Conference Call Service is not permitted on parties in the bridged state and may also be more restrictive if the principal of the bridge has an analog station or the exclusion option is in effect from a station associated with the bridge or PCOL.

Syntax

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

// cstaConferenceCall() - Service Request

RetCode_t         cstaConferenceCall (
    ACSHandle_t       acsHandle,
    InvokeID_t        invokeID,
    ConnectionID_t    *heldCall,        // devIDType= STATIC_ID
    ConnectionID_t    *activeCall,      // devIDType= STATIC_ID
    PrivateData_t     *privateData);

// CSTAConferenceCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t       acsHandle;
    EventClass_t      eventClass;// CSTACONFIRMATION
    EventType_t       eventType; // CSTA_CONFERENCE_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t      eventHeader;
    union
    {
        struct
        {
            InvokeID_t   invokeID;
            union
            {
            CSTAConferenceCallConfEvent_t      conferenceCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct Connection_t {
    ConnectionID_t party;
    DeviceID_t   staticDevice;         // NULL for not present
} Connection_t;

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

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

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTConferenceCallConfEvent - Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;  // ATT_CONFERENCE_CALL_CONF
    union
    {
        ATTConferenceCallConfEvent_t    conferenceCall;
    }u;
} ATTEvent_t;

typedef struct ATTConferenceCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTConferenceCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Consultation Call Service

Summary

- Direction: Client to Switch
- Function: *cstaConsultationCall()*
- Confirmation Event: *CSTAConsultationCallConfEvent*
- Private Data Function: *attV6ConsultationCall()* (private data version 6), *attConsultationCall()* (private data version 2, 3, 4, and 5)
- Private Data Confirmation Event: *ATTConsultationCallConfEvent* (private data version 5)
- Service Parameters: *activeCall, calledDevice*
- Private Parameters: *destRoute, priorityCalling, userInfo*
- Ack Parameters: *newCall*
- Ack Private Parameters: *ucid*
- Nak Parameter: *universalFailure*

Functional Description:

The Consultation Call Service places an existing active call (activeCall) at a device on hold and initiates a new call (newCall) from the same controlling device. This service provides the compound action of the Hold Call Service followed by Make Call Service. The Consultation Call service has the important special property of associating the Communication Manager Original Call Information from the call being placed on hold with the call being originated. This allows an application running at the consultation desktop to pop a screen using information associated with the call placed on hold. This is an important operation in call centers where an agent calls a specialist for consultation about a call in progress.

The Consultation Call Service request is acknowledged (Ack) by the switch if the switch is able to put the activeCall on hold and initiate a new call.

The request is negatively acknowledged if the switch:

- fails to put activeCall on hold (for example, call is in alerting state), or
- fails to initiate a new call (for example, invalid parameter).

If the request is negatively acknowledged, the TSAPI Service will attempt to put the activeCall to its original state, if the original state is known by the TSAPI Service before the service request. If the original state is unknown, there is no recovery for the activeCall's original state.

Service Parameters:

activeCall [mandatory] A valid connection identifier that indicates the connection to be placed on hold. This party must be in the active (talking) state or already held. The device associated with the activeCall must be a station. If the party specified in the request refers to a trunk device, the request will be denied. The deviceID in activeCall must contain the station extension of the controlling device.

calledDevice [mandatory] Must be a valid on-PBX extension or off-PBX number. On-PBX extension may be a station extension, VDN, split, hunt group, announcement extension, or logical agent's login ID. The calledDevice may include TAC/ARS/AAR information for off-PBX numbers. Trunk Access Code, Authorization Codes, and Force Entry of Account Codes can be specified with the calledDevice as if they were entered from the voice terminal using the keypad.

Private Parameters

destruct [optional] Specifies the TAC/ARS/AAR information for an off- PBX destination, if the information is not included in the calledDevice. A NULL indicates this parameter is not specified.

priority Calling [mandatory] Specifies the priority of the call. Values are On (TRUE) or Off (FALSE). If On is selected, a priority call is attempted for an on-PBX calledDevice. Note that Communication Manager does not permit priority calls to certain types of extensions (such as VDNs).

userInfo [optional] Contains user-to-user information. This parameter allows an application to associate caller information, up to 32 or 96 bytes, with a call. This information may be a customer number, credit card number, alphanumeric digits, or a binary string.

It is propagated with the call whether the call is made to a destination on the local switch or to a destination on a remote switch over PRI trunks. The switch sends the UUI in the ISDN SETUP message over the PRI trunk to establish the call. The local and the remote switch include the UUI in the Delivered Event Report and in the cstaRouteRequestEvent to the application. A NULL indicates this parameter is not present.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes.

Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE - There is no data provided in the data parameter.
- UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

Ack Parameters:

newCall [mandatory] A connection identifier indicates the connection between the controlling device and the new call. The newCall parameter contains the callID of the call and the station extension of the controlling device.

Ack Private Parameters:

ucid [optional] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- GENERIC_UNSPECIFIED (0) The specified data provided for the userInfo parameter exceeds the maximum allowable size. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. See the description of the userInfo parameter.
- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in activeCall.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) The connection identifier contained in the request is invalid or does not correspond to a station.
- NO_ACTIVE_CALL (24) The party to be put on hold is not currently active (for example, in alerting state) so it cannot be put on hold.
- GENERIC_STATE_INCOMPATIBILITY (21) (CS0/18) The originator does not go off-hook within five seconds after originating the call and cannot be forced off-hook.
- RESOURCE_BUSY (33) The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, etc.) to the same device.
- OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44) The client attempted to put a third party (two parties are on hold already) on hold on an analog station.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in activeCall.

Detailed Information:

See Detailed Information: in the “Hold Call Service” section and Detailed Information: on page 273.

Syntax

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

// cstaConsultationCall() - Service Request

RetCode_t         cstaConsultationCall (
    ACSHandle_t       acsHandle,
    InvokeID_t        invokeID,
    ConnectionID_t    *activeCall,      // devIDType= STATIC_ID
    DeviceID_t        *calledDevice,
    PrivateData_t     *privateData);

// CSTAConsultationCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_CONSULTATION_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAConsultationCallConfEvent_t consultationCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAConsultationCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAConsultationCallConfEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6ConsultationCall() - Service Request Private Data
                            // Setup Function

RetCode_t    attV6ConsultationCall(
    ATTPrivateData_t*privateData,
    DeviceID_t      *destRoute, // NULL indicates not specified
    Boolean         priorityCalling; // TRUE = On, FALSE = Off
    ATTUserToUserInfo_t *userInfo);  // NULL indicates not
                                     // specified

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef struct  ATTUserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
        short length;   // 0 indicates UUI not present
        unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE         = -1,  // indicates not specified
    UUI_USER_SPECIFIC = 0,  // user-specific
    UUI_IA5_ASCII    = 4    // null-terminated ascii
                            // character string
} ATTUUIProtocolType_t;
```

Private Data Version 6 Syntax (Continued)

```
// ATTConsultationCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;// ATT_CONSULTATION_CALL_CONF
    union
    {
    ATTConsultationCallConfEvent_t  consultationCall;
    }u;
} ATTEvent_t;

typedef struct ATTConsultationCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTConsultationCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attConsultationCall() - Service Request Private Data
                                  // Setup Function

RetCode_t    attConsultationCall(
     ATTPrivateData_t   *privateData,
     DeviceID_t         *destRoute,   // NULL indicates not specified
     Boolean            priorityCalling; // TRUE = On, FALSE = Off
     ATTUserToUserInfo_t *userInfo);   // NULL indicates not
                                       // specified

typedef struct ATTPrivateData_t {
     char      vendor[32];
     ushort    length;
     char      data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct     ATTV5UserToUserInfo_t  {
     ATTUUIProtocolType_t type;
     struct {
          short length;      // 0 indicates UUI not present
          unsigned char value[33];
     } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
     UUI_NONE            = -1,    // indicates not specified
     UUI_USER_SPECIFIC   = 0,     // user-specific
     UUI_IA5_ASCII       = 4      // null-terminated ascii
                                  // character string
} ATTUUIProtocolType_t;

// ATTConsultationCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
     ATTEventType_t     eventType;// ATT_CONSULTATION_CALL_CONF
     union
     {
     ATTConsultationCallConfEvent_t       consultationCall;
     }u;
} ATTEvent_t;

typedef struct ATTConsultationCallConfEvent_t
{
     ATTUCID_t     ucid;
} ATTConsultationCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Consultation Direct-Agent Call Service

Summary

- Direction: Client to Switch
- Function: *cstaConsultationCall()*
- Confirmation Event: *CSTAConsultationCallConfEvent*
- Private Data Function: *attV6DirectAgentCall() (private data version 6), attDirectAgentCall() (private data version 2, 3, 4, and 5)*
- Private Data Confirmation Event: *attConsultationCallConfEvent*
- Service Parameters: *activeCall, calledDevice*
- Private Parameters: *split, priorityCalling, userInfo*
- Ack Parameters: *newCall*
- Ack Private Parameters: *ucid*
- Nak Parameter: *universalFailure*

Functional Description:

The Consultation Direct-Agent Call Service places an existing active call (activeCall) at a device on hold and initiates a new direct-agent call (newCall) from the same controlling device. This service provides the compound action of the Hold Call Service followed by Make Direct-Agent Call Service. Like the Consultation Service, the Consultation Direct Agent Call service has the important special property of associating the Communication Manager Original Call Information from the call being placed on hold with the call being originated. This allows an application running at the consultation desktop to pop a screen using information associated with the call placed on hold. This is an important operation in call centers where an agent calls a specialist for consultation about a call in progress.

The Consultation Direct-Agent Call Service request is acknowledged by the switch if the switch is able to put the activeCall on hold and initiates a new direct-agent call.

The request is negatively acknowledged if the switch:

- Fails to put activeCall on hold (for example, call is in alerting state), or
- Fails to initiate a new direct-agent call (for example, invalid parameter).

If the request is negatively acknowledged, the TSAPI Service will attempt to put the activeCall into the active state, if it was in the active or held state.

The Consultation Direct Agent Call Service should be used only in the following two situations:

- Consultation Direct Agent Calls in a non-EAS environment

- Consultation Direct Agent Calls in an EAS environment only when it is required to ensure that these calls against a skill other than that skill specified for these measurements on the DEFINITY PBX for that agent.

Preferably in an EAS environment, Consultation Direct Agent Calls can be made using the Make Call service and specifying an Agent login-ID as the destination device. In this case Consultation Direct Agent Calls will be measured against the skill specified or those measurements on the DEFINITY PBX for that agent.

Service Parameters:

activeCall	[mandatory] A valid connection identifier that indicates the connection to be placed on hold. This party must be in the active (talking) state or already held. The device associated with the activeCall must be a station. If the party specified in the request refers to a trunk device, the request will be denied. The deviceID in activeCall must contain the station extension of the controlling device.
calledDevice	[mandatory] Must be a valid ACD agent extension. Agent at calledDevice must be logged in.

Private Parameters:

split	[mandatory] Contains a valid split extension. Agent at calledDevice must be logged into this split.
priorityCalling	[mandatory] Specifies the priority of the call. Values are On (TRUE) or Off (FALSE). If On is selected, a priority call is attempted for an on-PBX calledDevice. Note that Communication Manager does not permit priority calls to certain types of extensions (such as VDNs).
userInfo	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string. It is propagated with the call whether the call is made to a destination on the local switch or to a destination on a remote switch over PRI trunks. The switch sends the UUI in the ISDN SETUP message over the PRI trunk to establish the call. The local and the remote switch include the UUI in the Delivered Event Report and in the cstaRouteRequestEvent to the application. A NULL indicates this parameter is not present. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. **Note:** An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch. The following UUI protocol types are supported: ● UUI_NONE - There is no data provided in the data parameter. ● UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter. ● UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

Ack Parameters:

newCall	[mandatory] A connection identifier indicates the connection between the controlling device and the new call. The newCall parameter contains the callID of the call and the station extension of the controlling device.

Ack Private Parameters:

ucid	[optional] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- GENERIC_UNSPECIFIED (0) The specified data provided for the userInfo parameter exceeds the maximum allowable size. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. See userInfo on page 271.
- GENERIC_UNSPECIFIED (0) (CS3/11, CS3/15) Agent is not a member of the split or agent is not currently logged in split.
- VALUE_OUT_OF_RANGE (3) (CS0/100, CS0/96) The split contains an invalid value or invalid information element contents was detected.
- INVALID_CALLING_DEVICE (5) (CS3/27) The callingDevice is out of service or not administered correctly in the switch.
- PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS0/21, CS0/52) The COR check for completing the call failed. The call was attempted over a trunk that the originator has restricted from use.
- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in activeCall, the calledDevice is an invalid station extension, or the split does not contain a valid hunt group extension.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) The connection identifier contained in the request is invalid or does not correspond to a station.
- INVALID_DESTINATION (14) (CS3/24) The call was answered by an answering machine.
- INVALID_OBJECT_TYPE (18) (CS0/58, CS3/80) There is incompatible bearer service for the originating or destination address. For example. the originator is administered as a data hotline station or the destination is a data station. Call options are incompatible with this service.
- GENERIC_STATE_INCOMPATIBILITY (21) (CS0/18) The originator does not go off-hook within five seconds after originating the call and cannot be forced off-hook.
- INVALID_OBJECT_STATE (22) (CS0/98) Request (message) is incompatible with call state
- NO_ACTIVE_CALL (24) The party to be put on hold is not currently active (for example, in alerting state) so it cannot be put on hold.
- RESOURCE_BUSY (33) (CS0/17) The user is busy on another call and cannot originate this call. The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) to the same device.

- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILTY (41) (CS0/50) Service or option not subscribed/provisioned (AMD must be enabled).
- OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44) The client attempted to put a third party (two parties are on hold already) on hold on an analog station.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in activeCall.

Detailed Information:

See Detailed Information: in the "Hold Call Service" section and Programming details: in the "Make Direct-Agent Call Service" section in this chapter.

Syntax

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

// cstaConsultationCall() - Service Request

RetCode_t         cstaConsultationCall (
    ACSHandle_t       acsHandle,
    InvokeID_t        invokeID,
    ConnectionID_t    *activeCall,      // devIDType= STATIC_ID
    DeviceID_t        *calledDevice,
    PrivateData_t     *privateData);

// CSTAConsultationCallConfEvent - Service Response

typedef struct
{
    EventType_t     eventType; // CSTA_CONSULTATION_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAConsultationCallConfEvent_t consultationCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAConsultationCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAConsultationCallConfEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6DirectAgentCall() - Service Request Private Data
                            // Setup Function

RetCode_t    attV6DirectAgentCall(
    ATTPrivateData_t*privateData,
    DeviceID_t      *split,     // NULL indicates not specified
    Boolean         priorityCalling;// TRUE = On, FALSE = Off
    ATTUserToUserInfo_t *userInfo); // NULL indicates not
                                // specified

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef struct  ATTUserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
        short length;       // 0 indicates UUI not present
        unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE         = -1,  // indicates not specified
    UUI_USER_SPECIFIC = 0,  // user-specific
    UUI_IA5_ASCII    = 4    // null-terminated ascii
                            // character string
} ATTUUIProtocolType_t;
```

Private Data Version 6 Syntax (Continued)

```
// ATTConsultationCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;// ATT_CONSULTATION_CALL_CONF
    union
    {
    ATTConsultationCallConfEvent_t  consultationCall;
    }u;
} ATTEvent_t;

typedef struct ATTConsultationCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTConsultationCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attDirectAgentCall() - Service Request Private Data
                                  // Setup Function

RetCode_t    attDirectAgentCall(
     ATTPrivateData_t   *privateData,
     DeviceID_t         *split,       // NULL indicates not specified
     Boolean            priorityCalling;// TRUE = On, FALSE = Off
     ATTUserToUserInfo_t *userInfo);  // NULL indicates not
                                      // specified

typedef struct ATTPrivateData_t {
     char     vendor[32];
     ushort   length;
     char     data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct     ATTV5UserToUserInfo_t  {
     ATTUUIProtocolType_t type;
     struct {
          short length;          // 0 indicates UUI not present
          unsigned char value[33];
     } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
     UUI_NONE            = -1,    // indicates not specified
     UUI_USER_SPECIFIC   = 0,     // user-specific
     UUI_IA5_ASCII       = 4      // null-terminated ascii
                                  // character string
} ATTUUIProtocolType_t;

// ATTConsultationCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
     ATTEventType_t     eventType;// ATT_CONSULTATION_CALL_CONF
     union
     {
     ATTConsultationCallConfEvent_t       consultationCall;
     }u;
} ATTEvent_t;

typedef struct ATTConsultationCallConfEvent_t
{
     ATTUCID_t     ucid;
} ATTConsultationCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Consultation Supervisor-Assist Call Service

Summary

- Direction: Client to Switch
- Function: *cstaConsultationCall()*
- Confirmation Event: *CSTAConsultationCallConfEvent*
- Private Data Function: *attV6SupervisorAssistCall()* (private data version 6), *attSupervisorAssistCall()* (private data version 2, 3, 4, and 5)
- Private Data Confirmation Event: *attConsultationCallConfEvent*
- Service Parameters: *activeCall, calledDevice*
- Private Parameters: *split, userInfo*
- Ack Parameters: *newCall*
- Ack Private Parameters: *ucid*
- Nak Parameter: *universalFailure*

Functional Description:

The Consultation Supervisor-Assist Call Service places an existing active call (activeCall) at a device on hold and initiates a new supervisor-assist call (newCall) from the same controlling device. This service provides the compound action of the Hold Call Service followed by Make Supervisor-Assist. Like the Consultation Service, the Consultation Supervisor-Assist Call service has the important special property of associating the Communication Manager Original Call Information from the call being placed on hold with the call being originated. This allows an application running at the consultation desktop to pop a screen using information associated with the call placed on hold. This is an important operation in call centers where an agent calls a specialist for consultation about a call in progress.

The Consultation Supervisor-Assist Call Service request is acknowledged (Ack) by the switch if the switch is able to put the activeCall on hold and initiates a new supervisor-assist call.

The request is negatively acknowledged if the switch:

- Fails to put activeCall on hold (for example, call is in alerting state), or
- Fails to initiate a new direct-agent call (for example, invalid parameter).

If the request is negatively acknowledged, the TSAPI Service will attempt to put the activeCall into the active state, if it was in the active or held state.

Service Parameters:

activeCall	[mandatory] A valid connection identifier that indicates the connection to be placed on hold. This party must be in the active (talking) state or already held. The device associated with the activeCall must be a station. If the party specified in the request refers to a trunk device, the request will be denied. The deviceID in activeCall must contain the station extension of the controlling device.
calledDevice	[mandatory] Must be a valid ACD agent extension. Agent at calledDevice must be logged in.

Private Parameters:

split	[mandatory] Contains a valid split extension. Agent at calledDevice must be logged into this split.
userInfo	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string. It is propagated with the call whether the call is made to a destination on the local switch or to a destination on a remote switch over PRI trunks. The switch sends the UUI in the ISDN SETUP message over the PRI trunk to establish the call. The local and the remote switch include the UUI in the Delivered Event Report and in the cstaRouteRequestEvent to the application. A NULL indicates this parameter is not present. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. **Note:** An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch. The following UUI protocol types are supported: ● UUI_NONE - There is no data provided in the data parameter. ● UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter. ● UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

Ack Parameters:

newCall	[mandatory] A connection identifier indicates the connection between the controlling device and the new call. The newCall parameter contains the callID of the call and the station extension of the controlling device.

Ack Private Parameters:

ucid [optional] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- GENERIC_UNSPECIFIED (0) The specified data provided for the userInfo parameter exceeds the maximum allowable size. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. See the description of the userInfo parameter.
- GENERIC_UNSPECIFIED (0) (CS3/11, CS3/15) The agent is not a member of the split or the agent is not currently logged in split.
- VALUE_OUT_OF_RANGE (3) (CS0/100, CS0/96) The split contains an invalid value or invalid information element contents was detected.
- INVALID_CALLING_DEVICE (5) (CS3/27) The callingDevice is out of service or not administered correctly in the switch.
- PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS0/21, CS0/52) The COR check for completing the call failed. The call was attempted over a trunk that the originator has restricted from use.
- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in activeCall, the calledDevice is an invalid station extension, or the split does not contain a valid hunt group extension.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) The connection identifier contained in the request is invalid or does not correspond to a station.
- INVALID_DESTINATION (14) (CS3/24) The call was answered by an answering machine.
- INVALID_OBJECT_TYPE (18) (CS0/58, CS3/80) There is incompatible bearer service for the originating or destination address. For example, the originator is administered as a data hotline station or the destination is a data station. Call options are incompatible with this service.
- GENERIC_STATE_INCOMPATIBILITY (21) (CS0/18) The originator does not go off-hook within five seconds after originating the call and cannot be forced off-hook.

- INVALID_OBJECT_STATE (22) (CS0/98) Request (message) is incompatible with the call state.
- NO_ACTIVE_CALL (24) The party to be put on hold is not currently active (for example, in alerting state) so it cannot be put on hold.
- RESOURCE_BUSY (33) (CS0/17) The user is busy on another call and cannot originate this call. The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) to the same device.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILTY (41) (CS0/50) Service or option not subscribed/provisioned (AMD must be enabled).
- OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44) The client attempted to put a third party on hold on an analog station when two parties are already on hold.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in activeCall.

Detailed Information:

See Detailed Information: in the "Hold Call Service" section and Programming details: in the "Make Direct-Agent Call Service" section in this chapter.

Syntax

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

// cstaConsultationCall() - Service Request

RetCode_t         cstaConsultationCall (
    ACSHandle_t       acsHandle,
    InvokeID_t        invokeID,
    ConnectionID_t    *activeCall,      // devIDType= STATIC_ID
    DeviceID_t        *calledDevice,
    PrivateData_t     *privateData);

// CSTAConsultationCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_CONSULTATION_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAConsultationCallConfEvent_t consultationCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAConsultationCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAConsultationCallConfEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6SupervisorAssistCall() - Service Request Private Data
                            // Setup Function

RetCode_t    attV6SupervisorAssistCall(
    ATTPrivateData_t*privateData,
    DeviceID_t      *split,     // mandatory
                                // NULL indicates not specified
    ATTUserToUserInfo_t *userInfo); // NULL indicates not
                                // specified

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef struct  ATTUserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
                short length; // 0 indicates UUI not present
                unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE         = -1,  // indicates not specified
    UUI_USER_SPECIFIC = 0,  // user-specific
    UUI_IA5_ASCII    = 4    // null-terminated ascii
                            // character string
} ATTUUIProtocolType_t;
```

Private Data Version 6 Syntax (Continued)

```
// ATTConsultationCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;// ATT_CONSULTATION_CALL_CONF
    union
    {
    ATTConsultationCallConfEvent_t  consultationCall;
    }u;
} ATTEvent_t;

typedef struct ATTConsultationCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTConsultationCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSupervisorAssistCall() - Service Request Private Data
                                 // Setup Function

RetCode_t    attSupervisorAssistCall(
     ATTPrivateData_t   *privateData,
     DeviceID_t         *split,       // mandatory
                                      // NULL indicates not specified
     ATTUserToUserInfo_t *userInfo);  // NULL indicates not
                                      // specified

typedef struct ATTPrivateData_t {
     char     vendor[32];
     ushort   length;
     char     data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct     ATTV5UserToUserInfo_t  {
     ATTUUIProtocolType_t type;
     struct {
                short length; // 0 indicates UUI not present
                unsigned char value[33];
     } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
     UUI_NONE            = -1,    // indicates not specified
     UUI_USER_SPECIFIC   = 0,     // user-specific
     UUI_IA5_ASCII       = 4      // null-terminated ascii
                                  // character string
} ATTUUIProtocolType_t;

// ATTConsultationCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
     ATTEventType_t     eventType;// ATT_CONSULTATION_CALL_CONF
     union
     {
     ATTConsultationCallConfEvent_t       consultationCall;
     }u;
} ATTEvent_t;

typedef struct ATTConsultationCallConfEvent_t
{
     ATTUCID_t     ucid;
} ATTConsultationCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Deflect Call Service

Summary

- Direction: Client to Switch
- Function: *cstaDeflectCall ()*
- Confirmation Event: *CSTADeflectCallConfEvent*
- Service Parameters: *deflectCall, calledDevice*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

This service redirects an alerting call at a device to a new destination, either on-PBX or off-PBX. The call at the redirecting device is dropped after a successful redirection. An application may redirect an alerting call (at different devices) any number of times until the call is answered or dropped by the caller.

The service request is positively acknowledged if the call has successfully redirected for an on-PBX destination. For an off-PBX destination, this does not imply a successful redirection. It indicates that the switch attempted to redirect the call to the off-PBX destination and subsequent call progress events or tones may indicate redirection success or failure.

If the service request is negatively acknowledged, the call remains at the redirecting device and the calledDevice is not involved in the call.

Service Parameters:

deflectCall	[mandatory] Specifies the connectionID of the call that is to be redirected to another destination. The call must be in the alerting state at the device. The device must be a valid voice station extension.
calledDevice	[mandatory] Specifies the destination to which the call is redirected. The destination can be an on-PBX or off-PBX endpoint. For on-PBX endpoints, the calledDevice may be stations, queues, announcements, VDNs, or logical agent extensions.

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS3/42)
  - Attempted to redirect a call back to the call originator or to the redirecting device itself.
  - Attempted to redirect a call on the calledDevice of a cstaMakePredictiveCall.
- INVALID_OBJECT_STATE (22) (3/63)
  - An invalid callID or device identifier is specified in deflectCall.
  - The deflectCall is not in alerting state.
  - Attempted to redirect the call while in vector processing.
- PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE (8) (CS3/43)
  The request may fail because of one of the following:
  - invalid destination specified
  - toll restrictions on destination
  - COR restrictions on destination
  - destination is remote access extension
  - call origination restriction on the redirecting device
  - call is in vector processing
- RESOURCE_BUSY (33) (CS0/17) A call redirected to a busy station, a station that has call forwarding active, or a TEG group with one or more members busy will be rejected with this error.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) (CS0/50) This service is requested on a DEFINITY Server administered as a release earlier than G3V4.
- GENERIC_OPERATION (1) (CS0/111) This service is requested on a queued call or protocol error in the request.

Detailed Information:

- Administration Without Hardware - A call cannot be redirected to/from an AWOH station. However, if the AWOH station is forwarded to a real physical station, the call can be redirected to/from such a station, if it is being alerted.
- Attendants - Calls on attendants cannot be redirected.
- Auto Call Back - ACB calls cannot be redirected by the cstaDeflectCall service from the call originator.
- Bridged Call Appearance - A call may be redirected away from a primary extension or from a bridged station. When that happens, the call is redirected away from the primary and all bridged stations.

- Call Waiting - A call may be redirected while waiting at a busy analog set.
- Deflect From Queue - This service will not redirect a call from a queue to a new destination.
- Delivered Event - If the calling device or call is monitored, an application subsequently receives Delivered (or Network Reached) Event when redirection succeeds.
- Diverted Event - If the redirecting device is monitored by a cstaMonitorDevice() or the call is monitored by a cstaMonitorCallsViaDevice(), it will receive a Diverted Event when the call is successfully redirected, but there will be no Diverted Event for a cstaMonitorCall association.
- Loop Back - A call cannot be redirected to the call originator or to the redirecting device itself.
- Off-PBX Destination - If the call is redirected to an off-PBX destination, the caller will hear call progress tones. There may be conditions (for example, trunk not available) that will prevent the call from being placed. The call is nevertheless routed in those cases, and the caller receives busy or reorder treatment. An application may subsequently receive Failed, Call Cleared, or Connection Cleared Events if redirection fails.

  If trunk-to-trunk transfer is disallowed by the switch administration, redirection of an incoming trunk call to an off-PBX destination will fail.
- Priority and Forwarded Calls - Priority and forwarded calls are allowed to be redirected with cstaDeflectCall.
- Service Availability- This service is only available on a Communication Manager with G3V4 or later software.

Syntax

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

// cstaDeflectCall() - Service Request

RetCode_t         cstaDeflectCall (
    ACSHandle_t       acsHandle,
    InvokeID_t        invokeID,
    ConnectionID_t    *deflectCall,
    DeviceID_t        *calledDevice,
    PrivateData_t     *privateData);

// CSTADeflectCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_DEFLECT_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTADeflectCallConfEvent_t      deflectCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTADeflectCallConfEvent_t {
    Nulltype    null;
} CSTADeflectCallConfEvent_t;
```

Hold Call Service

Summary

- Direction: Client to Switch
- Function: *cstaHoldCall ()*
- Confirmation Event: *CSTAHoldCallConfEvent*
- Service Parameters: *activeCall, reservation*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

The Hold Call Service places a call on hold at a PBX station. The effect is as if the specified party depressed the hold button on his or her multifunction station to locally place the call on hold or switch-hook flashed on an analog station.

Service Parameters:

activeCall	[mandatory] A valid connection identifier that indicates the connection to be placed on hold. This party must be in the active (talking) state or already held. The device associated with the activeCall must be a station. If the party specified in the request refers to a trunk device, the request will be denied. The deviceID in activeCall must contain the station extension of the controlling device.
reservation	[optional - not supported] Specifies whether the facility is reserved for reuse by the held call. Communication Manager always allows a party to reconnect to a held call. It is recommended that the application always supply TRUE. In actuality, the TSAPI Service ignored the application-supplied value for this parameter.

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in activeCall.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) The connection identifier contained in the request is invalid or does not correspond to a station.
- NO_ACTIVE_CALL (24) The party to be put on hold is not currently active (for example, in alerting state) so it cannot be put on hold.
- RESOURCE_BUSY (33) The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, etc.) to the same device.
- OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44) The client attempted to put a third party on hold (two parties are on hold already) on an analog station.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in activeCall.

Detailed Information:

- Analog Stations - An analog station can not switch between a soft-held call and an active call from the voice set. However, with Hold Call Service, this is possible by placing the active call on hard-hold and retrieving the soft-held call. Hold Call Service places a call on conference and/or transfer hold. If that device already had a conference and/or transfer held call and a Hold Call Service is requested, the active call will be placed on hard-hold (unless there is call-waiting, in which case the request is denied).

  Note:
  A maximum of two calls may be in a held state at the same time. A request to have a third call on hold on the same analog station will be denied.

- Bridged Call Appearance - Hold Call Service is not permitted on parties in the bridged state and may also be more restrictive if the principal of the bridge has an analog station or the exclusion option is in effect from a station associated with the bridge or PCOL.
- Busy Verification of Terminals - A Hold Call Service request will be denied if requested for the verifying user's station.
- Held State - If the party is already on hold on the specified call when the switch receives the request, a positive request acknowledgment is returned.
- Music on Hold - Music on Hold (if administered and available) will be given to a party placed on hold from the other end either manually or via the Hold Call Service.

- Switch Operation - After a party is placed on hold through a Hold Call Service request, the user will not receive dial tone regardless of the type of phone device. Thus, subsequent calls must be placed by selecting an idle call appearance or through the Make Call Service request.

Syntax

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

// cstaHoldCall() - Service Request

RetCode_t       cstaHoldCall (
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    ConnectionID_t  *activeCall, // devIDType = STATIC_ID
    Boolean         reservation, // not supported - defaults to On
    DeviceID_t      *calledDevice,
    PrivateData_t   *privateData);

// CSTAHoldCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_HOLD_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAHoldCallConfEvent_t         holdCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAHoldCallConfEvent_t {
    Nulltype    null;
} CSTAHoldCallConfEvent_t;
```

Make Call Service

Summary

- Direction: Client to Switch
- Function: *cstaMakeCall()*
- Confirmation Event: *CSTAMakeCallConfEvent*
- Private Data Function: *attV6MakeCall()* (private data version 6), *attMakeCall()* (private data version 2, 3, 4, and 5)
- Private Data Confirmation Event: *ATTMakeCallConfEvent*
- Service Parameters: *callingDevice, calledDevice*
- Private Parameters: *destRoute, priorityCalling, userInfo*
- Ack Parameters: *newCall*
- Ack Private Parameters: *ucid*
- Nak Parameter: *universalFailure*

Functional Description:

The Make Call Service originates a call between two devices. The service attempts to create a new call and establish a connection with the originating device (callingDevice). The Make Call Service also provides a connection identifier (newCall) that indicates the connection of the originating device in the CSTAMakeCallConfEvent.

The client application uses this service to set up a call on behalf of a station extension (calling party) to either an on- or off-PBX endpoint (calledDevice). This service can be used by many types of applications such as Office Automation, Messaging, and Outbound Call Management (OCM) for Preview Dialing.

All trunk types (including ISDN-PRI) are supported as facilities for reaching called endpoints for outbound cstaMakeCall calls. Call progress feedback is reported as events to the application via Monitor Services. Answer Supervision or Call Classifier is not used for this service.

For the originator to place the call, the callingDevice (display or voice) must have an available call appearance for call origination and must not be in the talking (active) state on any call appearances. The originator is allowed to have a call(s) on hold or alerting at the device.

For a digital voice terminal without a speakerphone, when the switch selects the available call appearance for call origination, the red and green status lamps of the call appearance will light. The originator must go off-hook within five seconds. If the call is placed for an analog station without a speakerphone (or a handset), the user must either be idle or off-hook with dial tone, or go off-hook within five seconds after the Make Call request. In either case, the request will be denied if the station fails to go off-hook within five seconds.

The originator may go off-hook and receive dial tone first, and then issue the Make Call Service request for that station. The switch will originate the call on the same call appearance and callID to establish the call.

If the originator is off-hook busy, the call cannot be placed and the request is denied (RESOURCE_BUSY). If the originator is unable to originate for other reasons (see the Nak parameter universalFailure), the switch denies the request.

Service Parameters:

\

callingDevice [mandatory] Must be a valid station extension or, for phantom calls, an AWOH (administered without hardware) station extension.

Note:
For DEFINITY ECS switch software Release 6.3 and later, a call can be originated from an AWOH station or some group extensions (that is, a plain [non-ACD] hunt group). This is termed a *phantom call*. Most calls that can be requested for a physical extension can also be requested for an AWOH station and the associated event will also be received. If the call is made on behalf of a group extension, this may not apply. For more information, see “Phantom Call,” in the *Avaya MultiVantage Application Enablement Services, Release 3.1, ASAI Technical Reference*, Issue 2, 03-300549.

calledDevice [mandatory] Must be a valid on-PBX extension or off-PBX number. On-PBX extension may be a station extension, VDN, split, hunt group, announcement extension, or logical agent’s login ID. The calledDevice may include TAC/ARS/AAR information for off-PBX numbers. Trunk Access Code, Authorization Codes, and Force Entry of Account Codes can be specified with the calledDevice as if they were entered from the voice terminal using the keypad.

Private Parameters:

destRoute [optional] Specifies the TAC/ARS/AAR information for an off- PBX destination, if the information is not included in the calledDevice. A NULL indicates that this parameter is not specified.

priorityCalling [mandatory] Specifies the priority of the call. Values are "On" (TRUE) or "Off" (FALSE). If On is selected, a priority call is attempted for an on-PBX calledDevice. Note that Communication Manager does not permit priority calls to certain types of extensions (such as VDNs).

userInfo [optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string.

It is propagated with the call whether the call is made to a destination on the local switch or to a destination on a remote switch over PRI trunks. The switch sends the UUI in the ISDN SETUP message over the PRI trunk to establish the call. The local and the remote switch include the UUI in the Delivered Event Report and in the cstaRouteRequestEvent to the application. A NULL indicates this parameter is not present.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes.

Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE - There is no data provided in the data parameter.
- UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

Ack Parameters:

newCall [mandatory] A connection identifier that indicates the connection between the originating device and the call. The newCall parameter contains the callID of the call and the station extension of the callingDevice.

Ack Private Parameters:

ucid [optional] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure A Make Call request will be denied if the request fails before the call is attempted by the PBX.

If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- GENERIC_UNSPECIFIED (0) The specified data provided for the userInfo parameter exceeds the maximum allowable size. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. See the description of the userInfo parameter.
- INVALID_CALLING_DEVICE (5) The callingDevice is out of service or not administered correctly in the switch.
- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in callingDevice.
- GENERIC_STATE_INCOMPATIBILITY (21) The originator does not go off-hook within five seconds after originating the call and cannot be forced off-hook.
- RESOURCE_BUSY (33) The user is busy on another call and cannot originate this call, or the switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) to the same device.

Programming details:

- VDN - Priority calls cannot be made to VDNs. Do not set priorityCalling to TRUE when the calledDevice is a VDN.
- AAR/ARS - The AAR/ARS features are accessible by an application through Make Call Service. The calledDevice may include TAC/ARS/AAR information for off-PBX numbers (the switch uses only the first 32 digits as the number). However, it is recommended that, in situations where multiple applications (TSAPI applications and other applications) use ARS trunks, ARS Routing Plans be administered using partitioning to guarantee use of certain trunks to the Telephony Services API application. Each partition should be dedicated to a particular application (this is enforced by the switch).
  - If the application wants to obtain trunk availability information when ARS/AAR is used (in the calledDevice), it must query the switch about all trunk groups in the ARS partition dedicated. The application may not use the ARS/AAR code in the query to obtain trunk availability information.
  - When using ARS/AAR, the switch does not tell the application which particular trunk group was selected for a given call.
  - Care must be given to the proper administration of this feature, particularly the FRLs. If these are not properly assigned, calls may be denied despite trunk availability.
  - The switch does not attempt to validate the ARS/AAR code prior to placing the call.

- ARS must be subscribed in Communication Manager if outbound calls are made over ISDN-PRI facilities.

- ACD Destination - When the destination is an agent login ID or an ACD split, ACD call delivery rules apply. If an ACD agent's extension is specified in the calledDevice, the call is delivered to that ACD agent as a personal call, not a direct agent call.
- ACD Originator - Make Call Service cannot have an ACD Split as the callingDevice.
- Analog Stations - A maximum of three calls (one soft-held, one hard-held, and one active[1]) may be present at the same time at an analog station. In addition, the station may have a call waiting call.
  - A request to have more than three calls present will be denied. For example, if an analog station user has three calls present and another call waiting, the user cannot place the active call on hold or answer the call. The only operations allowed are drop the active call or transfer/conference the soft-held and active waiting call.
- Announcement Destination - Announcement calledDevices are treated like on-PBX station users.
- Attendants - The attendant group is not supported with Make Call Service. It may never be specified as the callingDevice and in some cases cannot be the calledDevice.
- Authorization Codes - If applicable, the originator will be prompted for authorization codes through the phone. The access codes and authorization codes can also be included in the calledDevice, if applicable, as if they were entered from the originator's voice terminal.
- Bridged Call Appearance - Make Call Service will always originate the call at the primary extension number of a user having a bridged appearance. For a call to originate at the bridged call appearance of a primary extension, that user must be off-hook at that bridged appearance at the time the Make Call Service is requested.
- Call Classification - All call-progress audible tones are provided to the originating user at the calling device (except that the user does not hear dial tone or touch tones). For OCM preview dialing applications, final call classification is done by the station user staffing the callingDevice (who hears call progress tones and manually records the result). If the call was placed to a VDN extension, the originator will hear whatever has been programmed for the vector associated with that VDN.
- Call Coverage Path Containing VDNs - Make Call Service is permitted to follow the VDN in the coverage path, provided that the coverage criteria has been met.
- Call Destination - If the calledDevice is on-PBX station, the user at the station will receive alerting. The user is alerted according to the call type (ACD or normal). Call delivery depends on the call type, station type, station administered options (manual/auto answer, call waiting, etc.), and station's talk state.

1. An active party/connection/call is a party/connection/call at the connected state. The user of an active party/connection/call usually has an active talk path and is talking or listening on the call.

  - For example, for an ACD call, if the user is off-hook idle, and in auto-answer mode, the call is cut-through immediately. If the user is off-hook busy and has a multifunction-function set, the call will alert a free appearance. If the user is off-hook busy and has an analog set, and the user has "call waiting", the analog station user is given the "call waiting tone". If the user is off-hook busy on an analog station and does not have "call waiting", the calling endpoint will hear busy. If the user is off-hook, alerting is started.

- Call Forwarding All Calls - A Make Call Service to a station (calledDevice) with the Call Forwarding All Calls feature active will redirect to the "forwarded to" station.
- Class of Restrictions (COR) - The Make Call Service is originated by using the originator's COR. A call placed to a called endpoint whose COR does not allow the call to end will return intercept treatment to the calling endpoint and the Failed Event Report with the error PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9).
- Class of Service (COS) - The Class of Service for the callingDevice is never checked for the Make Call Service.
- Data Calls - Data calls cannot be originated via the Make Call Service.
- DCS - A call made by Make Call Service over a DCS network is treated as an off-PBX call.
- Display - If the callingDevice has a display set, the display will show the extension and name of the calledDevice, if the calledDevice is on-PBX, or the name of the trunk group, if the calledDevice is off-PBX. If the calledDevice is on-PBX, normal display interactions apply for calledDevice with displays.
- Forced Entry of Account Codes - Make Call Service request attempted to trunk groups with the Forced Entry of Account Codes feature assigned which is allowed. It is up to the user at the callingDevice to enter the account codes via the touch-tone pad. Account code may not be provided via the TSAPI. If the originator of such a call is logged into an adjunct-controlled split (and therefore has the voice set locked), such a user will be unable to enter the required codes and will eventually get denial treatment.
- Hot Line - A Make Call Service request made on behalf of a station that has the Hot Line feature administered will be denied.
- Last Number Dialed - The calledDevice in a Make Call Service request is the last number dialed for the calledDevice until the next call origination from the callingDevice. Therefore, the user can use the "last number dialed" button to originate a call to the destination provided in the last Make Call Service request.
- Logical Agents - The callingDevice may contain a logical agent's login ID or a logical agent's physical station. If a logical agent's login ID is specified and the logical agent is logged in, the call is originated from the agent's station extension associated with the agent's login ID. If a logical agent's login ID is specified and the logical agent is not logged in, the call is denied with error INVALID_CALLING_DEVICE.
  - If the calledDevice contains a logical agent's login ID, the call is originated as if the call had been dialed from the callingDevice to the requested login ID. If the callingDevice and the calledDevice CORs permit, the call is treated as a direct agent call; otherwise, the call is treated as a personal call to the requested agent.

- Night Service - Make Call Service to splits in night service will go to night service.
- Personal Central Office Line (PCOL) - For a Make Call Service request originated at the PCOL call appearance of a primary extension, that user must be off-hook on the PCOL call appearance at the time the service is requested.
- PRI - An outgoing call over a PRI facility provides call feedback events from the network.
- Priority Calling - The user can originate a priority call by going off-hook, dialing the feature access code for priority calling, and requesting a Make Call Service.
- Send All Calls (SAC) - Make Call Service can be requested for a station (callingDevice) that has SAC activated. SAC has no effect on the callingDevice for the cstaMakeCall request.
- Single-Digit Dialing - Make Service request accepts single-digit dialing (for example, 0 for operator).
- Skill Hunt Groups - Make Call Service cannot have a skill hunt group extension as the callingDevice.
- Station Message Detail Recording (SMDR) - Calls originated by an application via the Make Call Service are marked with the condition code "B".
- Switch Operation - Once the call is successfully originated, the switch will not drop it regardless of outcome. The only exception is the denial outcome, which results in the intercept tone being played for 30 seconds after the call is disconnected. The originating station user or application drops cstaMakeCall calls either by going on-hook or via CSTA call control services. For example, if the application places a call to a busy destination, the originator will be busy until he/she normally drops or until the application sends a Clear Call or Clear Connection Service to drop the call.
- Terminating Extension Group (TEG) - Make Call Service requests cannot have the TEG group extension as the callingDevice. TEGs can only receive calls, not originate them.
- VDN - VDN cannot be the callingDevice of a Make Call Service, but it can be the calledDevice.
- VDN Destination - When the calledDevice is a VDN extension, vector processing rules apply.

Syntax

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

// cstaMakeCall() - Service Request

RetCode_t       cstaMakeCall (
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    DeviceID_t      *callingDevice,
    DeviceID_t      *calledDevice,
    PrivateData_t   *privateData);

// CSTAMakeCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_HOLD_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAMakeCallConfEvent_t         makeCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAMakeCallConfEvent_t {
    Nulltype    null;
} CSTAMakeCallConfEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6MakeCall() - Service Request Private Data Setup Function

RetCode_t    attV6MakeCall(
    ATTPrivateData_t*privateData,
    DeviceID_t      *destRoute, // NULL indicates not specified
    Boolean         priorityCalling,// TRUE = On, FALSE = Off
    ATTUserToUserInfo_t *userInfo); // NULL indicates not specified

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef struct  ATTUserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
                short length; // 0 indicates UUI not present
                unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE         = -1,  // indicates not specified
    UUI_USER_SPECIFIC = 0,  // user-specific
    UUI_IA5_ASCII    = 4    // null-terminated ascii
                            // character string
} ATTUUIProtocolType_t;
```

Private Data Version 6 Syntax (Continued)

```
// ATTMakeCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;// ATT_MAKE_CALL_CONF
    union
    {
    ATTMakeCallConfEvent_t          makeCall;
    }u;
} ATTEvent_t;

typedef struct ATTMakeCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTMakeCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMakeCall() - Service Request Private Data Setup Function

RetCode_t    attMakeCall(
     ATTPrivateData_t   *privateData,
     DeviceID_t         *split,       // NULL indicates not specified
     ATTUserToUserInfo_t *userInfo);  // NULL indicates not
                                      // specified

typedef struct ATTPrivateData_t {
     char     vendor[32];
     ushort   length;
     char     data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct     ATTV5UserToUserInfo_t  {
     ATTUUIProtocolType_t type;
     struct {
                  short length; // 0 indicates UUI not present
                  unsigned char value[33];
     } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
     UUI_NONE            = -1,    // indicates not specified
     UUI_USER_SPECIFIC   = 0,     // user-specific
     UUI_IA5_ASCII       = 4      // null-terminated ascii
                                  // character string
} ATTUUIProtocolType_t;

// ATTMakeCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
     ATTEventType_t     eventType;// ATT_MAKE_CALL_CONF
     union
     {
     ATTMakeCallConfEvent_t               makeCall;
     }u;
} ATTEvent_t;

typedef struct ATTMakeCallConfEvent_t
{
     ATTUCID_t     ucid;
} ATTMakeCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Make Direct-Agent Call Service

Summary

- Direction: Client to Switch
- Function: *cstaMakeCall()*
- Confirmation Event: *CSTAMakeCallConfEvent*
- Private Data Function: *attV6DirectAgentCall()* (private data version 6), *attDirectAgentCall()* (private data version 2, 3, 4, and 5)
- Private Data Confirmation Event: *ATTMakeCallConfEvent*
- Service Parameters: *callingDevice, calledDevice*
- Private Parameters: *split, priorityCalling, userInfo*
- Ack Parameters: *newCall*
- Ack Private Parameters: *ucid*
- Nak Parameter: *universalFailure*

Functional Description:

Make Direct-Agent Call Service is a special type of Make Call Service. Make Direct-Agent Call Service originates a call between two devices: a user station and an ACD agent logged into a specified split. The service attempts to create a new call and establish a connection with the originating device (callingDevice) first. The Direct-Agent Call service also provides a CSTA connection Identifier (newCall) that indicates the connection of the originating device in the CSTAMakeCallConfEvent.

This type of call may be used by applications whenever the application decides that the call originator should talk to a specific ACD agent. The application must specify the split extension (via database lookup) to which the calledDevice (ACD agent) is logged in. Direct-Agent calls can be tracked by Call Management Service (CMS) through the split measurements.

Service Parameters:

callingDevice [mandatory] Must be a valid station extension or an AWOH station extension (for phantom calls).

Note: For DEFINITY ECS switch software Release 6.3 and later, a call can be originated from an AWOH station or some group extensions (that is, a plain [non-ACD] hunt group). This is termed a phantom call. Most calls that can be requested for a physical extension can also be requested for an AWOH station and the associated event will also be received. If the call is made on behalf of a group extension, this may not apply. For more information, see "Phantom Call," in the *Avaya MultiVantage Application Enablement Services, Release 3.1, ASAI Technical Reference*, Issue 2, 03-300549.

This parameter may contain a logical agent's login ID (Logical Direct-Agent Call) or an agent's physical station extension. If the callingDevice contains a logical agent's login ID and the logical agent is logged in, the direct-agent call is originated from the agent's station. If the callingDevice contains a logical agent's login ID and the logical agent is not logged in, the direct-agent call is denied. The Logical Direct-Agent Call is only available when the Expert Agent Selection (EAS) feature is enabled on Communication Manager.

calledDevice [mandatory] Must be a valid ACD agent extension. Agent at calledDevice must be logged in. If calledDevice is a logical agent's ID, it is already treated by DEFINITY as a direct agent call and, in this case, private data should not be used. Doing so would result in error INVALID_CSTA_DEVICE_IDENTIFIER (12).

Private Parameters:

split [mandatory] Contains a valid split extension. Agent at calledDevice must be logged into this split.

priorityCalling [mandatory] Specifies the priority of the call. Values are On (TRUE) or Off (FALSE). If On is selected, a priority call is attempted for an on-PBX calledDevice. Note that Communication Manager does not permit priority calls to certain types of extensions (such as VDNs).

userInfo [optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string.

It is propagated with the call. The switch sends the UUI in the Delivered Event Report to the application. A NULL indicates that this parameter is not present.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes.

Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE - There is no data provided in the data parameter.
- UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

Ack Parameters:

newCall [mandatory] A connection identifier that indicates the connection between the originating device and the call. The newCall parameter contains the callID of the call and the station extension of the callingDevice.

Ack Private Parameters:

ucid [optional] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure A Make Call request will be denied if the request fails before the call is attempted by the PBX.

If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- GENERIC_UNSPECIFIED (0) The specified data provided for the userInfo parameter exceeds the maximum allowable size. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. See the description of the userInfo parameter.
- GENERIC_UNSPECIFIED (0) (CS3/11, CS3/15) Agent is not a member of the split or agent is not currently logged in split.
- VALUE_OUT_OF_RANGE (3) (CS0/100, CS0/96) The split contains an invalid value or invalid information element contents was detected.
- INVALID_CALLING_DEVICE (5) (CS3/27) The callingDevice is out of service or not administered correctly in the switch.
- PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS0/21, CS0/52) The COR check for completing the call failed. The call was attempted over a trunk that the originator has restricted from use.
- INVALID_DESTINATION (14) (CS3/24) The call was answered by an answering machine.
- INVALID_OBJECT_STATE (22) (CS0/98) Request (message) is incompatible with call state.
- INVALID_CSTA_DEVICE_IDENTIFIER (12) (CS0/28) The split does not contain a valid hunt group extension. The callingDevice or calledDevice is an invalid station extension.
- INVALID_OBJECT_TYPE (18) (CS0/58, CS3/80) There is an incompatible bearer service for the originating or destination address (for example, the originator is administered as a data hotline station or the destination is a data station).
- Call options are incompatible with this service.

- GENERIC_STATE_INCOMPATIBILITY (21) (CS0/18) The originator does not go off-hook within five seconds after originating the call and cannot be forced off-hook.
- RESOURCE_BUSY (33) (CS0/17) The user is busy on another call and cannot originate this call. The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) to the same device.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILTY (41) (CS0/50) Service or option not subscribed/provisioned (AMD must be enabled).

Programming details:

See also, Programming details: on page 252 for related information about the Make Call Service.

- Display - If the calledDevice has a display set, it will show the specified split's name and extension. If the destination ACD agent has a display, it will show the name of the originator and the name of the specified split.
- Logical Agents - The callingDevice may contain a logical agent's login ID or a logical agent's physical station. If a logical agent's login ID is specified and the logical agent is logged in, the call originates from the agent's station extension associated with the agent's login ID. If a logical agent's login ID is specified and the logical agent is not logged in, the call is denied with the error INVALID_CALLING_DEVICE.

Syntax

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

// cstaMakeCall() - Service Request

RetCode_t       cstaMakeCall (
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    DeviceID_t      *callingDevice,
    DeviceID_t      *calledDevice,
    PrivateData_t   *privateData);

// CSTAMakeCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_MAKE_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAMakeCallConfEvent_t         makeCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAMakeCallConfEvent_t
{
    ConnectionID_t newCall; // devIDType = STATIC_ID
} CSTAMakeCallConfEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6DirectAgentCall() - Service Request Private Data
                            // Setup Function

RetCode_t    attV6DirectAgentCall(
    ATTPrivateData_t*privateData,
    DeviceID_t      *split,     // mandatory
                                // NULL indicates not specified
    Boolean         priorityCalling;// TRUE = On, FALSE = Off
    ATTUserToUserInfo_t *userInfo); // NULL indicates not
                                // specified

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef struct  ATTUserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
        short length;       // 0 indicates UUI not present
        unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE         = -1,  // indicates not specified
    UUI_USER_SPECIFIC = 0,  // user-specific
    UUI_IA5_ASCII    = 4    // null-terminated ascii
                            // character string
} ATTUUIProtocolType_t;
```

Private Data Version 6 Syntax (Continued)

```
// ATTMakeCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;// ATT_MAKE_CALL_CONF
    union
    {
    ATTMakeCallConfEvent_t          makeCall;
    }u;
} ATTEvent_t;

typedef struct ATTMakeCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTMakeCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attDirectAgentCall() - Service Request Private Data
                                  // Setup Function

RetCode_t    attDirectAgentCall(
     ATTPrivateData_t   *privateData,
     DeviceID_t         *split,       // mandatory
                                      // NULL indicates not specified
     Boolean            priorityCalling;   // TRUE = On, FALSE = Off
     ATTUserToUserInfo_t *userInfo);  // NULL indicates not
                                      // specified

typedef struct ATTPrivateData_t {
     char     vendor[32];
     ushort   length;
     char     data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct     ATTV5UserToUserInfo_t  {
     ATTUUIProtocolType_t type;
     struct {
          short length;          // 0 indicates UUI not present
          unsigned char value[33];
     } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
     UUI_NONE            = -1,    // indicates not specified
     UUI_USER_SPECIFIC   = 0,     // user-specific
     UUI_IA5_ASCII       = 4      // null-terminated ascii
                                  // character string
} ATTUUIProtocolType_t;

// ATTMakeCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
     ATTEventType_t     eventType;// ATT_MAKE_CALL_CONF
     union
     {
     ATTMakeCallConfEvent_t               makeCall;
     }u;
} ATTEvent_t;

typedef struct ATTMakeCallConfEvent_t
{
     ATTUCID_t     ucid;
} ATTMakeCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Make Predictive Call Service

Summary

- Direction: Client to Switch
- Function: *cstaMakePredictiveCall()*
- Confirmation Event: *CSTAMakePredictiveCallConfEvent*
- Private Data Function: *attV6MakePredictiveCall() (private data version 6), attMakePredictiveCall()* (private data version 2, 3, 4, and 5)
- Private Data Confirmation Event: *ATTMakePredictiveCallConfEvent*
- Service Parameters: *callingDevice, calledDevice, allocationState*
- Private Parameters: *priorityCalling, maxRings, answerTreat, destRoute, userInfo*
- Ack Parameters: *newCall*
- Ack Private Parameters: *ucid*
- Nak Parameter: *universalFailure*

Functional Description:

The Make Predictive Call Service originates a Switch-Classified call between two devices. The service attempts to create a new call and establish a connection with the terminating (called) device first. The Make Predictive Call service also provides a CSTA Connection Identifier that indicates the connection of the terminating device. The call will be dropped if the call is not answered after the maximum ring cycle has expired. When Communication Manager is administered to return a classification, the classification appears in the Established event.

Predictive dial calls cannot use TAC dialing to either access trunks or to make outbound calls - TAC dialing will be blocked by the DEFINITY switch.

Service Parameters:

callingDevice	[mandatory] Must be a valid local extension number associated with an ACD split, hunt group, or announcement, a VDN in an EAS environment, or an AWOH station extension (for phantom calls). **Note:** For DEFINITY ECS switch software Release 6.3 and later, a call can be originated from an AWOH station or some group extensions (that is, a plain [non-ACD] hunt group). This is termed a phantom call. Most calls that can be requested for a physical extension can also be requested for an AWOH station and the associated event will also be received. If the call is made on behalf of a group extension, this may not apply. For more information, see "Phantom Call," in the *Avaya MultiVantage Application Enablement Services, Release 3.1, ASAI Technical Reference*, Issue 2, 03-300549.
calledDevice	[mandatory] Must be a valid on-PBX extension or off-PBX number. On-PBX extension must be a station extension. The calledDevice may include ARS/AAR information for off-PBX numbers. Authorization Codes and Force Entry of Account Codes can be specified with the calledDevice as if they were entered from the voice terminal using the keypad.
allocationState	[optional - partially supported] Specifies the condition in which the call attempts to connect to the caller (callingDevice). Only AS_CALL_ESTABLISHED is supported, meaning that Communication Manager will attempt to connect the call to the callingDevice if connected state is determined at the calledDevice. If AS_CALL_DELIVERED is specified, it will be ignored and default to AS_CALL_ESTABLISHED.

Private Parameters:

priorityCalling	[mandatory] Specifies the priority of the call. Values are On (TRUE) or Off (FALSE). If On is selected, a priority call is attempted for an on-PBX calledDevice. Note that Communication Manager does not permit priority calls to certain types of extensions (such as VDNs).
maxRings	[optional] Specifies the number of rings that are allowed before classifying the call as no answer. The minimum is two; the maximum is 15. If an out-of-range value is specified, it defaults to 10.
answerTreat	[mandatory] Specifies the call treatment when an answering machine is detected. ● AT_NONE - Treatment follows the switch answering machine detection administration. ● AT_DROP - Drops the call if an answering machine is detected. ● AT_CONNECT - Connects the call if an answering machine is detected. ● AT_NO_TREATMENT - Indicates that no answering machine treatment is specified.

destRoute [optional] Specifies the TAC/ARS/AAR information for off-PBX destinations if the information is not included in the calledDevice. A NULL indicates that this parameter is not specified.

userInfo [optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string.

It is propagated with the call whether the call is made to a destination on the local switch or to a destination on a remote switch over PRI trunks. The switch sends the UUI in the ISDN SETUP message over the PRI trunk to establish the call. The local and the remote switch include the UUI in the Delivered Event Report and in the cstaRouteRequestEvent to the application. A NULL indicates that this parameter is not present.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes.

Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE - There is no data provided in the data parameter.
- UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

Ack Parameters:

newCall [mandatory] A connection identifier that indicates the connection between the originating device and the call. The newCall parameter contains the callID of the call and the station extension of the callingDevice.

Ack Private Parameters:

ucid [optional] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure

A Make Call request will be denied if the request fails before the call is attempted by the PBX.

If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- GENERIC_UNSPECIFIED (0) The specified data provided for the userInfo parameter exceeds the maximum allowable size. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. See the description of the userInfo parameter.
- VALUE_OUT_OF_RANGE (3) (CS0/100, CS0/96) Invalid information element contents was detected.
- INVALID_CALLING_DEVICE (5) (CS3/27) The callingDevice is out of service or not administered correctly in the switch.
- PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS0/21, CS0/52) Attempted to use a Trunk Access Code (TAC) to access a PRI trunk (only AAR/ARS feature access codes may be used to place a switch-classified call over a PRI trunk). The COR check for completing the call failed. The call was attempted over a trunk that the originator has restricted from use.
- INVALID_CSTA_DEVICE_IDENTIFIER (12) (CS0/28) The callingDevice is neither a split nor an announcement extension.
- INVALID_OBJECT_TYPE (18) (CS0/58, CS3/80) There is incompatible bearer service for the originating or destination address. For example, the originator is administered as a data hotline station or the destination is a data station. Call options are incompatible with this service.
- INVALID_OBJECT_STATE (22) (CS0/98) Request (message) is incompatible with the call state.
- GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31) (CS3/22)
- One of the following conditions exists when switch attempted to make the call:
  - No Call classifier available
  - No time slot available
  - No trunk available
  - Queue full
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILTY (41) (CS0/50) Service or option not subscribed/provisioned. Answer Machine Detection is requested, but AMD is not enabled on the switch.

Detailed Information:

- The CSTAMakePredictiveCallConfEvent is sent to the application immediately after the switch accepts the CSTAMakePredictiveCall request and attempts to call the destination. The application receives a call ID in the CSTAMakePredictiveCallConfEvent. The application can monitor the outbound call and receives events of the call when the switch tries to connect the destination. When the outbound call is monitored, the call ID in the reported events remains unchanged when the destination answers and when the switch connects the calling device (normally this is a VDN or an ACD Split); that is, the call ID remains unchanged until the call is conferenced or transferred.
- The callingDevice and the calledDevice in the event reports resulting from the outbound call monitored by CSTAMonitorCall (using the call ID reported in the CSTAMakePredictiveCallConfEvent) are the same as those specified in the CSTAMakePredictiveCall request. However, this is different from the callingDevice and calledDevice in the events reported from the CSTAMonitorCallsViaDevice of the VDN/ACD Split or the cstaMonitorDevice() of the agent station. These monitors have an inbound call view instead of an outbound call view. Thus, the callingDevice is the calledDevice specified in the CSTAMakePredictiveCall request. The calledDevice is the callingDevice specified in the CSTAMakePredictiveCall request.
- If a client application wants to receive events for answering machine detection, the client application should establish a monitorCall, after the application receives a confirmation for the makePredictiveCall.
- For predictive calls whose source is a VDN that has a first step in its vector, an adjunct route request requires a CSTAMonitorCallsViaDevice to be placed on the VDN to guarantee correct UUI treatment when new UUI is entered in the route selection step. The applied monitor allows the UUI entered for the CSTAMakePredictiveCall request to show in the original call information UUI, while showing the UUI entered in the route select to show in the UUI field.

Note:

For predictive dial applications that are missing events because of race conditions, consider using enhanced VDN monitors. For more information, see Monitor Calls Via Device Service on page 462.

Syntax

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

// cstaMakePredictiveCall() - Service Request

RetCode_t       cstaMakePredictiveCall (
    ACSHandle_t         acsHandle,
    InvokeID_t          invokeID,
    DeviceID_t          *callingDevice,
    DeviceID_t          *calledDevice,
    AllocationState_t   allocationState,
    PrivateData_t       *privateData);

// CSTAMakePredictiveCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_MAKE_PREDICTIVE_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAMakePredictiveCallConfEvent_t      makePredictiveCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAMakePredictiveCallConfEvent_t
{
    ConnectionID_t newCall; // devIDType = STATIC_ID
} CSTAMakePredictiveCallConfEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6MakePredictiveCall() - Service Request Private Data
                            // Setup Function

RetCode_t    attV6MakePredictiveCall(
    ATTPrivateData_t*privateData,
    Boolean         priorityCalling;//TRUE = On, FALSE = Off
    short           maxRings,   // less than 2 or greater 15
                                    // are treated as not
                                    // specified
    ATTAnswerTreat_tanswerTreat,    // AT_NONE, AT_DROP, or
                                    // AT_CONNECT
    DeviceID_t      *destRoute,     // NULL = not specified
    ATTUserToUserInfo_t *userInfo); // NULL = not specified

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef struct  ATTUserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
            short length; // 0 indicates UUI not present
            unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef enum ATTUUIProtocolType_t {
    UUI_NONE         = -1,  // indicates not specified
    UUI_USER_SPECIFIC = 0,  // user-specific
    UUI_IA5_ASCII    = 4    // null-terminated ascii
                            // character string
} ATTUUIProtocolType_t;

typedef enum ATTAnswerTreat_t {
    AT_NO_TREATMENT= 0, // indicates treatment not specified
    AT_NONE      = 1,   // treatment follows machine instruct
    AT_DROP      = 2,   // drop call if machine detected
    AT_CONNECT   = 3    // connect call if machine detected


} ATTAnswerTreat_t;

// ATTMakePredictiveCallConfEvent -  Service Response
// Private Data
(supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;
                        // ATT_MAKE_PREDICTIVE_CALL_CONF
    union
    {
        ATTMakePredictiveCallConfEvent_tmakePredictiveCall;
    }u;
} ATTEvent_t;

typedef struct ATTMakePredictiveCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTMakePredictiveCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMakePredictiveCall() - Service Request Private Data
                            // Setup Function

RetCode_t    attMakePredictiveCall(
    ATTPrivateData_t*privateData,
    Boolean         priorityCalling;//TRUE = On, FALSE = Off
    short           maxRings,   // less than 2 or greater 15
                                    // are treated as not
                                    // specified
    ATTAnswerTreat_tanswerTreat,    // AT_NONE, AT_DROP, or
                                    // AT_CONNECT
    DeviceID_t      *destRoute,     // NULL = not specified
    ATTUserToUserInfo_t *userInfo); // NULL = not specified


typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct  ATTV5UserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
            short length; // 0 indicates UUI not present
            unsigned char value[33];
    } data;
} ATTV5UserToUserInfo_t;
```

Private Data Version 2-5 Syntax (Continued)

```
typedef enum ATTUUIProtocolType_t {
    UUI_NONE         = -1,  // indicates not specified
    UUI_USER_SPECIFIC = 0,  // user-specific
    UUI_IA5_ASCII    = 4    // null-terminated ascii
                            // character string
} ATTUUIProtocolType_t;

typedef enum ATTAnswerTreat_t {
    AT_NO_TREATMENT= 0, // indicates treatment not specified
    AT_NONE      = 1,   // treatment follows machine instruct
    AT_DROP      = 2,   // drop call if machine detected
    AT_CONNECT   = 3    // connect call if machine detected


} ATTAnswerTreat_t;

// ATTMakePredictiveCallConfEvent -  Service Response
// Private Data
(supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;
                        // ATT_MAKE_PREDICTIVE_CALL_CONF
    union
    {
        ATTMakePredictiveCallConfEvent_tmakePredictiveCall;
    }u;
} ATTEvent_t;

typedef struct ATTMakePredictiveCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTMakePredictiveCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Make Supervisor-Assist Call Service

Summary

- Direction: Client to Switch
- Function: *cstaMakeCall()*
- Confirmation Event: *CSTAMakeCallConfEvent*
- Private Data Function: *attV6SupervisorAssistCall()* (private data version 6), *attSupervisorAssistCall()* (private data version 2, 3, 4, and 5)
- Private Data Confirmation Event: *ATTMakeCallConfEvent*
- Service Parameters: *callingDevice, calledDevice*
- Private Parameters: *split, userInfo*
- Ack Parameters: *newCall*
- Ack Private Parameters: *ucid*
- Nak Parameter: *universalFailure*

Functional Description:

Make Supervisor-Assist Call Service is a special type of Make Call Service. This service originates a call between two devices: an ACD agent's extension and another station extension (typically a supervisor). The service attempts to create a new call and establish a connection with the originating (calling) device first. The Supervisor-Assist Call service also provides a CSTA Connection Identifier that indicates the connection of the originating device.

A call of this type is measured by CMS as a supervisor-assist call and is always a priority call.

This type of call is used by the application whenever an agent wants to consult with the supervisor. The agent must be logged into the specified ACD split to use this service.

Service Parameters:

callingDevice	[mandatory] Must be a valid ACD agent extension or an AWOH station extension (for phantom calls). Agent must be logged in. Note: For DEFINITY ECS switch software Release 6.3 and later, a call can be originated from an AWOH station or some group extensions (that is, a plain [non-ACD] hunt group). This is termed a phantom call. Most calls that can be requested for a physical extension can also be requested for an AWOH station and the associated event will also be received. If the call is made on behalf of a group extension, this may not apply. For more information, see "Phantom Call," in the *Avaya MultiVantage Application Enablement Services, Release 3.1, ASAI Technical Reference*, Issue 2, 03-300549.
calledDevice	[mandatory] Must be valid on-PBX station extension (excluding VDNs, splits, off-PBX DCS and UDP extensions).

Private Parameters:

split [mandatory] Specifies the ACD agent's split extension. The agent at callingDevice must be logged into this split.

userInfo [optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string. It is propagated with the call. The switch sends the UUI in the Delivered Event Report to the application. A NULL indicates that this parameter is not present.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes.

Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE - There is no data provided in the data parameter.
- UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

Ack Parameters:

newCall [mandatory] A connection identifier that indicates the connection between the originating device and the call. The newCall parameter contains the callID of the call and the station extension of the callingDevice.

Ack Private Parameters:

ucid [optional] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure A Make Call request will be denied if the request fails before the call is attempted by the PBX.

If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

GENERIC_UNSPECIFIED (0) The specified data provided for the userInfo parameter exceeds the maximum allowable size. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. See the description of the userInfo parameter.

GENERIC_UNSPECIFIED (0) (CS3/11, CS3/15) The agent is not a member of the split or the agent is not currently logged into the split.

VALUE_OUT_OF_RANGE (3) (CS0/100, CS0/96) The split contains an invalid value or invalid information element contents was detected.

INVALID_CALLING_DEVICE (5) (CS3/27) The callingDevice is out of service or not administered correctly in the switch.

PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS0/21, CS0/52) The COR check for completing the call failed. The call was attempted over a trunk that the originator has restricted from use.

INVALID_DESTINATION (14) (CS3/24) The call was answered by an answering machine.

INVALID_CSTA_DEVICE_IDENTIFIER (12) (CS0/28) The split does not contain a valid hunt group extension. The callingDevice or calledDevice is an invalid station extension.

INVALID_OBJECT_TYPE (18) (CS0/58, CS3/80) There is incompatible bearer service for the originating or destination address. For example, the originator is administered as a data hotline station or the destination is a data station. Call options are incompatible with this service.

GENERIC_STATE_INCOMPATIBILITY (21) (CS0/18) The originator does not go off-hook within five seconds after originating the call and cannot be forced off-hook.

INVALID_OBJECT_STATE (22) (CS0/98) Request (message) is incompatible with call state.

RESOURCE_BUSY (33) (CS0/17) The user is busy on another call and cannot originate this call. The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) to the same device.

GENERIC_SUBSCRIBED_RESOURCE_AVAILABILTY (41) (CS0/50) Service or option not subscribed/provisioned (AMD must be enabled).

Detailed Information:

See Programming details: in the "Make Call Service" section in this chapter.

Syntax

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

// cstaMakeCall() - Service Request

RetCode_t       cstaMakeCall (
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    DeviceID_t      *callingDevice,
    DeviceID_t      *calledDevice,
    PrivateData_t   *privateData);

// CSTAMakeCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_MAKE_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAMakeCallConfEvent_t         makeCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAMakeCallConfEvent_t
{
    ConnectionID_t newCall; // devIDType = STATIC_ID
} CSTAMakeCallConfEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6SupervisorAssistCall() - Service Request Private Data
                              // Setup Function

RetCode_t    attV6SupervisorAssistCall(
    ATTPrivateData_t*privateData,
    DeviceID_t      *split,     // mandatory
                                // NULL indicates not specified
    ATTUserToUserInfo_t *userInfo); // NULL indicates not
                                // specified

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef struct  ATTUserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
        short length;       // 0 indicates UUI not present
        unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE         = -1,  // indicates not specified
    UUI_USER_SPECIFIC = 0,  // user-specific
    UUI_IA5_ASCII    = 4    // null-terminated ascii
                            // character string
} ATTUUIProtocolType_t;
```

Private Data Version 6 Syntax (Continued)

```
// ATTMakeCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;// ATT_MAKE_CALL_CONF
    union
    {
    ATTMakeCallConfEvent_t          makeCall;
    }u;
} ATTEvent_t;

typedef struct ATTMakeCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTMakeCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSupervisorAssistCall() - Service Request Private Data
                             // Setup Function

RetCode_t    attSupervisorAssistCall(
    ATTPrivateData_t*privateData,
    DeviceID_t      *split,     // mandatory
                                // NULL indicates not specified
    ATTUserToUserInfo_t *userInfo); // NULL indicates not
                                // specified

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct  ATTV5UserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
        short length;       // 0 indicates UUI not present
        unsigned char value[33];
    } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE         = -1,  // indicates not specified
    UUI_USER_SPECIFIC = 0,  // user-specific
    UUI_IA5_ASCII    = 4    // null-terminated ascii
                            // character string
} ATTUUIProtocolType_t;
```

Private Data Version 2-5 Syntax (Continued)

```
// ATTMakeCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
    ATTEventType_t  eventType;// ATT_MAKE_CALL_CONF
    union
    {
    ATTMakeCallConfEvent_t          makeCall;
    }u;
} ATTEvent_t;

typedef struct ATTMakeCallConfEvent_t
{
    ATTUCID_t   ucid;
} ATTMakeCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Pickup Call Service

Summary

- Direction: Client to Switch
- Function: *cstaPickupCall ()*
- Confirmation Event: *CSTAPickupCallConfEvent*
- Service Parameters: *deflectCall, calledDevice*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

This service takes an alerting call at a device to another on-PBX device (within a DCS environment). The call at the alerting device is dropped after a successful redirection. An application may take an alerting call (at different devices) to another device any number of times until the call is answered or dropped by the caller.

The service request is positively acknowledged, if the call has successfully taken to another device.

If the service request is negatively acknowledged, the call remains at the alerting device and the calledDevice is not involved in the call.

Service Parameters:

deflectCall	[mandatory] Specifies the connectionID of the call that is to be taken to another destination. The call must be in alerting state at the device. The device must be a valid voice station extension.
calledDevice	[mandatory] Specifies the destination of the call. The destination must be an on-PBX endpoint. The calledDevice may be stations, queues, announcements, VDNs, or logical agent extension. Note that the calledDevice can be a device within a DCS environment.

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 .

- PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS3/42)
- Attempted to take a call back to the call originator or to the alerting device itself.
- Attempted to take a call on the calledDevice of a cstaMakePredictiveCall.
- INVALID_OBJECT_STATE (22) (3/63)
- An invalid callID or device identifier is specified in deflectCall.
- The deflectCall is not at alerting state.
- Attempted to take the call while in vector processing.
- INVALID_DESTINATION (14) (CS3/43) The request may fail because of one of the following:
  - Invalid destination specified
  - Toll restrictions on destination
  - COR restrictions on destination
  - Destination is remote access extension
  - Call origination restriction on the redirecting device
  - Call is in vector processing
- RESOURCE_BUSY (33) (CS0/17) The calledDevice is busy.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) (CS0/50) This service is requested on a switch administered as a release earlier than G3V4.
- GENERIC_OPERATION (1) (CS0/111) This service is requested on a queued call or protocol error in the request.

Detailed Information:

- Administration Without Hardware - A call cannot be redirected from/to an AWOH station. However, if the AWOH station is forwarded to a real physical station, the call can be redirected from/to such a station, if it is being alerted.
- Attendants - Calls on attendants cannot be redirected.
- Auto Call Back - ACB calls cannot be redirected by the cstaDeflectCall service from the call originator.
- Bridged Call Appearance - A call may be redirected away from a primary extension or from a bridged station. When that happens, the call is redirected away from the primary and all bridged stations.

- Call Forwarding, Cover All, Send All Call - Call redirection to a station with Call Forwarding/Cover All/Send All Call active can be picked up.
- Call Waiting - A call may be redirected while waiting at a busy analog set.
- cstaDeflectCall - The cstaPickupCall Service is similar to the cstaDeflectCall service, except that the calledDevice must be an on-PBX device. Note that the calledDevice can be a device within a DCS environment.
- Deflect From Queue - This service will not redirect a call from a queue to a new destination.
- Delivered Event - If the calling device or call is monitored, an application subsequently receives Delivered (or Network Reached) Event when redirection succeeds.
- Diverted Event - If the redirecting device is monitored by a cstaMonitorDevice() or the call is monitored by a cstaMonitorCallsViaDevice(), it will receive a Diverted Event when the call is successfully redirected, but there will be no Diverted Event for a cstaMonitorCall association.
- Loop Back - A call cannot be redirected back to call originator or to the redirecting device itself.
- Off-PBX Destination - If the call is redirected to an off-PBX destination, the caller will hear call progress tones. Some conditions (for example, trunk not available) may prevent the call from being placed. The call is nevertheless routed in those cases, and the caller receives busy or reorder treatment. An application may subsequently receive Failed, Call Cleared, Connection Cleared Events if redirection fails.

  If trunk-to-trunk transfer is disallowed by the switch administration, redirection of an incoming trunk call to an off-PBX destination will fail.
- Priority and Forwarded Calls - Priority and forwarded calls are allowed to be redirected with cstaDeflectCall.
- Service Availability - This service is only available on Communication Manager with G3V4 or later software.

Syntax

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

// cstaPickupCall() - Service Request

RetCode_t       cstaPickupCall (
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    ConnectionID_t  *deflectCall,
    DeviceID_t      *calledDevice,
    PrivateData_t   *privateData);

// CSTAPickupCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass;// CSTACONFIRMATION
    EventType_t     eventType; // CSTA_PICKUP_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
            CSTAPickupCallConfEvent_t       pickupCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAPickupCallConfEvent_t {
    Nulltype    null;
} CSTAPickupCallConfEvent_t;
```

Reconnect Call Service

Summary

- Direction: Client to Switch
- Function: cstaReconnectCall()
- Confirmation Event: CSTAReconnectCallConfEvent
- Private Data Function: attV6ReconnectCall() (private data version 6), attReconnectCall() (private data version 2, 3, 4, and 5)
- Service Parameters: activeCall, heldCall
- Private Parameters: dropResource, userInfo
- Ack Parameters: noData
- Nak Parameter: universalFailure

Functional Description:

The Reconnect Call Service allows a client to disconnect (drop) an existing connection from a call and then reconnect a previously held connection or answer an alerting (or bridged) call at the same device. It provides the compound action of the Clear Connection Service followed by a Retrieve Call Service or an Answer Call Service.

The Reconnect Call Service request is acknowledged (Ack) by the switch if the switch is able to retrieve the specified held heldCall or answer the specified alerting heldCall. The request is negatively acknowledged if switch fails to retrieve or answer heldCall.

The switch continues to retrieve or answer heldCall, even if it fails to drop activeCall.

Note:
A race condition may exist between human operation and the application request. The activeCall may be dropped before the service request is received by the switch. Since a station can have only one active call, the reconnect operation continues when the switch fails to drop the activeCall. If the activeCall cannot be dropped because a wrong connection is specified and there is another call active at the station, the retrieve heldCall operation will fail.

If the request is negatively acknowledged, the activeCall will not be in the active state, if it was in the active state.

Service Parameters:

activeCall [mandatory] A valid connection identifier that indicates the callID and the station extension (STATIC_ID). The deviceID in activeCall must contain the station extension of the controlling device. The local connection state of the call must be active.

heldCall [mandatory] A valid connection identifier that indicates the callID and the station extension (STATIC_ID). The deviceID in heldCall must contain the station extension of the controlling device. The local connection state of the call can be either alerting, bridged, or held.

Private Parameters:

dropResource [optional] Specifies the resource to be dropped from the call. The available resources are and DR_CALL_CLASSIFIER and DR_TONE_GENERATOR. The tone generator is any Communication Manager applied denial tone that is timed by the switch.

userInfo [optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string.

It is propagated with the call when the call is dropped and passed to the application in a Connection Cleared Event Report. A NULL indicates that this parameter is not present.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes.

Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE - There is no data provided in the data parameter.
- UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

Ack Parameter:

noData None for this service.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- GENERIC_UNSPECIFIED (0) The specified data provided for the userInfo parameter exceeds the maximum allowable size. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. See the description of the userInfo parameter.
- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in heldCall.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) An incorrect callID or an incorrect deviceID is specified in heldCall.
- GENERIC_STATE_INCOMPATIBILITY (21) The station user did not go off-hook for heldCall within five seconds and is not capable of being forced off-hook.
- INVALID_CONNECTION_ID_FOR_ACTIVE_CALL (23) The controlling deviceIDs in activeCall and heldCall are different.
- INVALID_OBJECT_STATE (22) The specified activeCall at the station is not currently active (in alerting or held state) so it cannot be dropped. The Reconnect Call Service operation stops and the heldCall will not be retrieved.
- The specified heldCall at the station is not in the alerting, connected, held, or bridged state.
- NO_CALL_TO_ANSWER (28) The call was redirected to coverage within the five-second interval.
- GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31) The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Clear Connection, etc.) to the same device.
- The client attempted to add a seventh party to a call with six active parties.
- RESOURCE_BUSY (33) User at the station is busy on a call or there are no idle appearances available.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in heldCall.

Detailed Information:

See the Detailed Information: in the "Answer Call Service" section, Detailed Information: in the "Clear Connection Service" section and Detailed Information: in the "Retrieve Call Service" section in this chapter.

Syntax

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

// cstaReconnectCall() - Service Request

RetCode_t         cstaReconnectCall (
     ACSHandle_t       acsHandle,
     InvokeID_t        invokeID,
     ConnectionID_t    *heldCall,        // devIDType= STATIC_ID
     ConnectionID_t    *activeCall,      // devIDType= STATIC_ID
     PrivateData_t     *privateData);

// CSTAReconnectCallConfEvent - Service Response

typedef struct
{
     ACSHandle_t       acsHandle;
     EventClass_t      eventClass;// CSTACONFIRMATION
     EventType_t       eventType; // CSTA_RECONNECT_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
     ACSEventHeader_t      eventHeader;
     union
     {
          struct
          {
               InvokeID_t   invokeID;
               union
               {
               CSTAReconnectCallConfEvent_t       reconnectCall;
               } u;
          } cstaConfirmation;
     } event;
     char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAReconnectCallConfEvent_t {
     Nulltype          null;
} CSTAReconnectCallConfEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6ReconnectCall() - Service Request Private Data
                        // Setup Function

RetCode_t    attV6ReconnectCall(
    ATTPrivateData_t    *privateData,
    ATTDropResource_t   dropResource);  // NULL indicates
                                        // no dropResource
                                        // specified
    ATTUserToUserInfo_t *userInfo);     // NULL indicates
                                        // no userInfo
                                        // specified

typedef struct ATTPrivateData_t {
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTDropResource_t {
    DR_NONE             = -1,// indicates not specified
    DR_CALL_CLASSIFIER  = 0,// call classifier to be dropped
    DR_TONE_GENERATOR   = 1 // tone generator to be dropped }
ATTDropResource_t;

#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef structATTUserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
        short length;       // 0 indicates UUI not present
        unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE            = -1,// indicates not specified
    UUI_USER_SPECIFIC   = 0, // user-specific
    UUI_IA5_ASCII       = 4  // null-terminated ascii
                             // character string
} ATTUUIProtocolType_t;
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attReconnectCall() - Service Request Private Data
                        // Setup Function

RetCode_t    attReconnectCall(
    ATTPrivateData_t    *privateData,
    ATTDropResource_t   dropResource);  // NULL indicates
                                        // no dropResource
                                        // specified
    ATTUserToUserInfo_t *userInfo);     // NULL indicates
                                        // no userInfo
                                        // specified

typedef struct ATTPrivateData_t {
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTDropResource_t {
    DR_NONE             = -1,// indicates not specified
    DR_CALL_CLASSIFIER  = 0,// call classifier to be dropped
    DR_TONE_GENERATOR   = 1 // tone generator to be dropped }
ATTDropResource_t;

typedef structATTV5UserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
        short length;       // 0 indicates UUI not present
        unsigned char value[33];
    } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
    UUI_NONE            = -1,// indicates not specified
    UUI_USER_SPECIFIC   = 0, // user-specific
    UUI_IA5_ASCII       = 4  // null-terminated ascii
                             // character string
} ATTUUIProtocolType_t;
```

Retrieve Call Service

Summary

- Direction: Client to Switch
- Function: *cstaRetrieveCall ()*
- Confirmation Event: *CSTARetrieveCallConfEvent*
- Service Parameters: *heldCall*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

The Retrieve Call Service connects an on-PBX held connection.

Service Parameters:

heldCall	[mandatory] A valid connection identifier that indicates the endpoint to be connected. The deviceID in heldCall must contain the station extension of the endpoint.

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in heldCall.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) The connectionID contained in the request is invalid.
- GENERIC_STATE_INCOMPATIBILITY (21) The user was on-hook when the request was made and he/she did not go off-hook within five seconds (call remains on hold).
- NO_ACTIVE_CALL (24) The specified call at the station is cleared so it cannot be retrieved.
- NO_HELD_CALL (25) The specified connection at the station is not in the held state (for example, alerting state) so it cannot be retrieved.
- RESOURCE_BUSY (33) The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Conference Call, etc.) to the same device.
- CONFERENCE_MEMBER_LIMIT_EXCEEDED (38) The client attempted to add a seventh party to a six-party conference call.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in heldCall.

Detailed Information:

- Active State - If the party is already retrieved on the specified call when the switch receives the request, a positive acknowledgment is returned.
- Bridged Call Appearance - Retrieve Call Service is not permitted on parties in the bridged state and may also be more restrictive if the principal of the bridge has an analog station or the exclusion option is in effect from a station associated with the bridge or PCOL.
- Hold State - Normally, the party to be retrieved has been placed on hold from the station or via the Hold Call Service.
- Switch Operation - A party may be retrieved only to the same call from which it had been put on hold as long as there is no other active call at the user's station.
  - If the user is on-hook (in the held state), the switch must be able to force the station off- hook or the user must go off-hook within five seconds after requesting a Retrieve Call Service. If one of the above conditions is not met, the request is denied (GENERIC_STATE_INCOMPATIBILITY) and the party remains held.

- If the user is listening to dial tone while a request for Retrieve Call Service is received, the dial tone will be dropped and the user reconnected to the held call.
- If the user is listening to any other kind of tone (for example, denial) or is busy talking on another call, the Retrieve Call Service request is denied (RESOURCE_BUSY).

Syntax

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

// cstaRetrieveCall() - Service Request

RetCode_t         cstaRetrieveCall (
     ACSHandle_t       acsHandle,
     InvokeID_t        invokeID,
     ConnectionID_t    *heldCall,        // devIDType= STATIC_ID
     PrivateData_t     *privateData);

// CSTARetrieveCallConfEvent - Service Response

typedef struct
{
     ACSHandle_t       acsHandle;
     EventClass_t      eventClass;// CSTACONFIRMATION
     EventType_t       eventType; // CSTA_RETRIEVE_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
     ACSEventHeader_t      eventHeader;
     union
     {
          struct
          {
               InvokeID_t   invokeID;
               union
               {
               CSTARetrieveCallConfEvent_t        retrieveCall;
               } u;
          } cstaConfirmation;
     } event;
     char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTARetrieveCallConfEvent_t {
     Nulltype          null;
} CSTARetrieveCallConfEvent_t;
```

Send DTMF Tone Service (Private Data Version 4 and Later)

Summary

- Direction: Client to Switch
- Function: *cstaEscapeService()*
- Confirmation Event: *CSTAEscapeServiceConfEvent*
- Private Data Function: *attSendDTMFToneExt()* (private data version 5 and later), *attSendDTMFTone()* (private data version 4)
- Service Parameters: *noData*
- Private Parameters: *sender, receivers, tones, toneDuration, pauseDuration*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

The Send DTMF Tone Service on behalf of an on-PBX endpoint sends a sequence of DTMF tones (maximum of 32) to endpoints on the call. The endpoints receiving the DTMF signal can be on-PBX or off-PBX. To send the DTMF tones, the call must be in an established state.

The allowed DTMF tones are digits 0-9 and # and *. Through such a tone sequence, an application could interact with far-end applications, such as automated bank tellers, automated attendants, voice mail systems, database systems, paging services, and so forth.

A CSTA Confirmation will be returned to the application when the service request has been accepted or when transmission of the DTMF tones has started. No event or indication will be provided to the application when the transmission of the DTMF tones is completed.

Service Parameters:

noData	None for this service.

Private Parameters

:

sender	[mandatory] Specifies the connectionID of the endpoint on whose behalf DTMF tones are to be sent. This connectionID can be an on-PBX endpoint or an off-PBX endpoint (via trunk connection) on the call.
receivers	[optional - not supported] A list of up to five connectionIDs that can receive the DTMF tones. If this list is empty (NULL or the count is 0), all parties on the call will receive the DTMF tones if eligible (that is, the voice path allows the party to receive the signals). This parameter is reserved for future use. If present, it will be ignored.
tones	[mandatory] DTMF sequence to be generated. The maximum tone sequence that can be sent is 32. The allowed DTMF tones are null-terminated ASCII string with digits 0-9, '#' and '*' only. Any other character in tones is invalid and will cause the request to be denied.
toneDuration	[optional] Specifies the number of one hundredth of a second (for example, 10 means 1/10 of a second) used to control the tone duration. The only valid values for the duration are 6 through 35 (one hundredths of a second).
pauseDuration	[optional] Specifies the number of one hundredth of a second used to control the pause duration. The only valid values are 4 through 10 (one hundredths of a second)

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 in Chapter 3:

- VALUE_OUT_OF_RANGE (3) (CS0/100) The tones parameter has length equal to 0 or greater than 32 or invalid characters are specified in tones. Also, could indicate that parameter values for either *toneDuration* or *pauseDuration* were incorrectly set.
- OBJECT_NOT_KNOWN (4) (CS0/96) Mandatory parameter is missing.
- INVALID_CSTA_DEVICE_IDENTIFIER (13) (CS0/28) Invalid deviceID is specified in sender.
- INVALID_OBJECT_STATE (22) (CS0/98, CS3/63) The service is requested on a call that is currently receiving switch-provided tone, such as dial tone, busy tone, ringback tone, intercept tone, Music-on-Hold/Delay, etc. The call must be in an established state in order to send DTMF tones.
- NO_ACTIVE_CALL (24) (CS3/86) Invalid callID is specified in sender or receivers.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) (CS0/50) This service is requested on a switch administered as a release earlier than G3V4.

Detailed Information:

- * And # Characters - If * and/or # characters are present, they will not be interpreted as termination characters or have any other transmission control function.
- AUDIX - AUDIX analog line ports connected to the Communication Manager will be able to receive DTMF tones generated by this service. However, embedded AUDIX or embedded AUDIX configured to emulate an analog line port interface is not supported.
- Call State - This service may be requested for any active call. This service will be denied when this feature is requested on a call that is currently receiving any switch-provided tone, such as busy, ringback, intercept, music-on-hold, etc.
- Connection State - A sender must have an active voice path to the call. A sender at alerting or held local state cannot send the DTMF tone. A receiver must have an active voice path to the sender. A receiver at hold local state will not receive the tone, although the switch will attempt to send the tone.
- DTMF Receiver - Only parties connected to the switch via analog line ports, analog trunk ports (including tie trunks), or digital trunk ports (including ISDN trunk ports) can be a receiver.

- DTMF Sender - Any voice station or (incoming) trunk caller on an active call can be a sender. DTMF tones will be sent to all parties (receivers) with proper connection type except the sender.
- Multiple Send DTMF Tone Requests - An application can send on behalf of different endpoints in a conference call such that DTMF tone sequences overlap or interfere with each other. An application is responsible for ensuring that it does not ask for multiple send DTMF tone requests from multiple parties on the same call at nearly the same time.
- Unsupported DTMF Tones - Tones corresponding to characters A, B, C, D are not supported by this service.
- Tone Cadence and Level - The application can only control the sequence of DTMF tones. The cadence and levels at which the tones are generated will be controlled by Communication Manager system administration and/or current defaults for the tone receiving ports, rather than by the application. When DTMF tones are sent to a multi-receiver call, the receivers may hear DTMF sequence with differing cadences.
- Service Availability - This service is only available on Communication Manager with G3V4 or later software.

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t         cstaEscapeService (
    ACSHandle_t       acsHandle,
    InvokeID_t        invokeID,
    PrivateData_t     *privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
    ACSHandle_t       acsHandle;
    EventClass_t      eventClass;// CSTACONFIRMATION
    EventType_t       eventType; // CSTA_ESCAPE_SVC_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t      eventHeader;
    union
    {
        struct
        {
            InvokeID_t    invokeID;
            union
            {
            CSTAEscapeSvcConfEvent_t            escapeService;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t {
    Nulltype          null;
} CSTAEscapeSvcConfEvent_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSendDTMFToneExt() - Service Request Private Data
                              // Setup Function

RetCode_t    attSendDTMFToneExt(
    ATTPrivateData_t  *privateData,
    ConnectionID_t    *sender;          // mandatory - NULL is
                                        // treated as not specified
    ATTConnIDList_t   *receivers;       // ignored - reserved for
                                        // future use (send to all
                                        // parties)
    char              *tones            // mandatory - NULL is
                                        // treated as not specified
    short             toneDuration,     // for tone duration
    short             pauseDuration);   // for pause duration


typedef struct ATTPrivateData_t {
    char     vendor[32];
    ushort   length;
    char     data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct    ATTConnIDList_t
{
    int               count;
    ConnectionID_t    *pParty;
} ATTConnIDList_t;
```

Private Data Version 4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSendDTMFTone() - Service Request Private Data
                              // Setup Function

RetCode_t    attSendDTMFTone(
    ATTPrivateData_t  *privateData,
    ConnectionID_t    *sender;          // mandatory - NULL is
                                        // treated as not specified
    ATTV4ConnIDList_t *receivers;       // ignored - reserved for
                                        // future use (send to all
                                        // parties)
    char              *tones            // mandatory - NULL is
                                        // treated as not specified
    short             toneDuration,     // ignored - reserved for
                                        // future use
    short             pauseDuration);   // ignored - reserved for
                                        // future use

typedef struct ATTPrivateData_t {
    char     vendor[32];
    ushort   length;
    char     data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct    ATTV4ConnIDList_t
{
    short             count;       // 0 means not specified
                                   // (send to all parties)
    ConnectionID_t    party[ATT_MAX_RECEIVERS];

} ATTV4ConnIDList_t;
```

Selective Listening Hold Service (Private Data Version 5 and Later)

Summary

- Direction: Client to Switch
- Function: *cstaEscapeService()*
- Confirmation Event: *CSTAEscapeServiceConfEvent*
- Private Data Function: *attSelectiveListeningHold()* (private data version 5 and later)
- Service Parameters: *noData*
- Private Parameters: *subjectConnection, allParties, selectedParty*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

The Selective Listening Hold Service allows a client application to prevent a specific party on a call from hearing anything said by another specific party or all other parties on the call. It allows a client application to put a party's (subjectConnection) listening path to a selected party (selectedParty) on listen-hold, or all parties on an active call on listen-hold. The selected party or all parties may be stations or trunks. A party that has been listen-held may continue to talk and be heard by other connected parties on the call since this service does not affect the talking or listening path of any other party. A party will be able to hear parties on the call from which it has not been listen-held, but will not be able to hear any party from which it has been listen-held. This service will also allow the listen-held party to be retrieved (i.e., to again hear the other party or parties on the call).

Service Parameters:

noData	None for this service.

Private Parameters:

subjectConnection	[mandatory] Specifies the connectionID of the party who will not hear the voice from all other parties or a single party specified in the selectedParty. This connectionID can be an on-PBX endpoint or an off-PBX endpoint (via trunk connection) on the call.
allParties	[mandatory] Specifies either all parties' or a single party's listening path is to be held from the subjectConnection party. True - the listening paths of all parties on the call will be held from the subjectConnection party. This prevents the subjectConnection from listening to all other parties on the call. The subjectConnection endpoint can still talk and be heard by all other connected parties on the call. The selectedParty parameter is ignored. False - the listening path of the subjectConnection party will be held from the selectedParty party. This prevents the subjectConnection from listening to all other parties on the call. The subjectConnection endpoint can still talk and be heard by all other connected parties on the call. The selectedParty parameter must be specified.
selectedParty	[optional] A connectionID whose voice will not be heard by the subjectConnection party. If allParties is false, a connectionID must be specified. If allParties is true, the connectionID in this parameter is ignored.

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786. • VALUE_OUT_OF_RANGE (3) (CS0/100) A party specified is not part of the call or in wrong state (e.g., a two-party call with the selectedParty still in the alerting state). • OBJECT_NOT_KNOWN (4) (CS0/96) Mandatory parameter is missing. • INVALID_CSTA_DEVICE_IDENTIFIER (13) (CS0/28) The party specified is not supported by this service (e.g., announcements, extensions without hardware, etc). • INVALID_OBJECT_STATE (22) (CS0/98) The request to listen-hold from all parties is not granted because there are no other eligible parties on the call (including any that were previously listen-held). • NO_ACTIVE_CALL (24) (CS3/63) Invalid callID is specified. • GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31) (CS3/40) Switch capacity has been exceeded. • GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) (CS0/50) This service has not been administratively enabled on the switch.

Detailed Information:

- Announcements - A party cannot be listen-held from an announcement. When a request is made to listen- hold all parties on a call, and there are more parties than just the announcement, the other parties will be listen-held, but the announcement will not. When the only other party on the call is an announcement, the request will fail.
- Attendants -This feature will not work with attendants.
- Call Vectoring - A call cannot be listen-held when in vector processing.
- Conference and Transfer Call - When two calls are conferenced/transferred, the listen-held state of one party (A) from another party (B) in the resulting call is determined as follows:
  - If party A was listen-held from party B in at least one of the original calls prior to the conference/transfer, party A will remain listen-held from party B in the resulting call.
  - Otherwise party A will not be listen-held from party B.

  When the request is received for a multi-party conference and one of the parties is still alerting, the request will be positively acknowledged and the alerting party will be listen-held upon answering.

- Converse Agent - A converse agent may be listen-held. While in this state, the converse agent will hear any DTMF digits that might be sent by the switch (as specified by the switch administration).
- DTMF Receiver - When a party has been listen-held while DTMF digits are being transmitted by the same switch (as a result of the Send DTMF service), the listen-held party will still hear the DTMF digits. However, the listen-held party will not hear the DTMF digits if the digits are sent by another switch.
- Hold Call - A party that is listen-held may be put on hold and retrieved as usual. A party that is already on hold and is being listen-held will be listen-held after having been retrieved. The service request on a held party will be positively acknowledged.
- Music On Hold - Music on Hold ports may not be listen-held (connection is not addressable). If a party is being listen-held from all other parties (while listening to Music on Hold), the party will still hear the Music on Hold.
- Park/Unpark Call - A call with parties listen-held may be parked. When the call is unparked, the listening paths that were previously held will remain on listen-hold.
- Retrieve Call - If a listen-held party goes on hold and then is retrieved, all listening paths that were listen-held will remain listen-held.
- Switch Administration - The Selective Listening Hold Service must be enabled (set to ‘y’) in order for it to work.

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t         cstaEscapeService (
     ACSHandle_t       acsHandle,
     InvokeID_t        invokeID,
     PrivateData_t     *privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
     ACSHandle_t       acsHandle;
     EventClass_t      eventClass;// CSTACONFIRMATION
     EventType_t       eventType; // CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
     ACSEventHeader_t      eventHeader;
     union
     {
          struct
          {
               InvokeID_t   invokeID;
          } cstaConfirmation;
     } event;
     char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSelectiveListeningHold() - Service Request Private
                                    // Data Setup Function

RetCode_t    attSelectiveListeningHold(
    ATTPrivateData_t*privateData,
    ConnectionID_t  *subjectConnection,
    Boolean         allParties;
    ConnectionID_t *selectedParty);

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// attSelectiveListeningHoldConfEvent - Private Data
                                        // Service Response

typedef struct ATTSelectiveListeningHoldConfEvent_t {       Nulltype
null;
} ATTSelectiveListeningHoldConfEvent_t;
```

Selective Listening Retrieve Service (Private Data Version 5 and Later)

Summary

- Direction: Client to Switch
- Function: *cstaEscapeService()*
- Confirmation Event: *CSTAEscapeServiceConfEvent*
- Private Data Function: *attSelectiveListeningRetrieve()*
- Service Parameters: *noData*
- Private Parameters: *subjectConnection, allParties, selectedParty*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

The Selective Listening Retrieve Service allows a client application to retrieve a party (subjectConnection) from listen-hold to another party (selectedParty) or all parties that were previously being listen-held.

Service Parameters:

noData	None for this service.

Private Parameters:

subjectConnection	[mandatory] Specifies the connectionID of the party whose listening path will be reconnected to all parties or party specified in the selectedParty. This connectionID can be an on-PBX endpoint or an off-PBX endpoint (via trunk connection) on the call.
allParties	[mandatory] Specifies either all parties' or a single party's listening path is to be reconnected from the subjectConnection party. True - the listening paths of all parties on the call will be reconnected from the subjectConnection party. This allows the subjectConnection endpoint to be able to listen to all other parties on the call. The selectedParty parameter is ignored. False - the listening path of the subjectConnection party will be reconnected from the selectedParty party. This allows the subjectConnection endpoint be able to listen to selectedParty party. The selectedParty parameter must be specified.
selectedParty	[optional] A connectionID whose listening path will be retrieved from listen-held by the subjectConnection party. If allParties is false, connectionIDs must be specified. If allParties is true, the connectionID in this parameter is ignored.

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- VALUE_OUT_OF_RANGE (3) (CS0/100) A party specified is not part of the call or in the wrong state (e.g., a two-party call with the selectedParty still in the alerting state).
- OBJECT_NOT_KNOWN (4) (CS0/96) Mandatory parameter is missing.
- INVALID_CSTA_DEVICE_IDENTIFIER (13) (CS0/28) The party specified is not supported by this feature (e.g., announcements, extensions without hardware, etc).
- NO_ACTIVE_CALL (24) (CS3/63) Invalid callID is specified.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) (CS0/50) This service has not been administratively enabled on the switch.

Detailed Information:

See Detailed Information: in the "Selective Listening Hold Service" section in this chapter for details.

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t         cstaEscapeService (
     ACSHandle_t       acsHandle,
     InvokeID_t        invokeID,
     PrivateData_t     *privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
     ACSHandle_t       acsHandle;
     EventClass_t      eventClass;// CSTACONFIRMATION
     EventType_t       eventType; // CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
     ACSEventHeader_t      eventHeader;
     union
     {
          struct
          {
               InvokeID_t   invokeID;
          } cstaConfirmation;
     } event;
     char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSelectiveListeningRetrieve() - Service Request Private
                                     // Data Setup Function

RetCode_t    attSelectiveListeningRetrieve(
    ATTPrivateData_t*privateData,
    ConnectionID_t  *subjectConnection,
    Boolean         allParties;
    ConnectionID_t *selectedParty);

typedef struct ATTPrivateData_t {
    char    vendor[32];
    ushort  length;
    char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// attSelectiveListeningRetrieveConfEvent - Private Data
                                          // Service Response

typedef struct ATTSelectiveListeningRetrieveConfEvent_t {
    Nulltype        null;                                    }
ATTSelectiveListeningRetrieveConfEvent_t;
```

Single Step Conference Call Service (Private Data Version 5 and Later)

Summary

- Direction: Client to Switch
- Function: *cstaEscapeService()*
- Confirmation Event: *CSTAEscapeServiceConfEvent*
- Private Data Function: *attSingleStepConferenceCall()*
- Private Data Confirmation Event: *ATTSingleStepConferenceCallConfEvent*
- Service Parameters: *noData*
- Private Parameters: *activeCall, deviceToBeJoin, participationType, alertDestination*
- Ack Parameters: *noData*
- Ack Private Parameters: *transferredCall*
- Nak Parameter: *universalFailure*

Functional Description:

The Single Step Conference Call Service will join a new device into an existing call. This service can be repeated to make n-device conference calls (subject to switching function limits). Currently DEFINITY supports six (6) parties on a call.

Note:
Single Step Conference Call Service is not currently supported by an ISDN BRI station.

Service Parameters:

noData	None for this service.

Private Parameters

activeCall [mandatory] A pointer to a connection identifier in the call to which a new device is to be added. This can be any connection on the call.

deviceToBeJoin [mandatory] A pointer to the device identifier that is to be added to the call. This must be either a physical station extension of any type or an extension administered without hardware (AWOH), but not a group extension.

Physical stations may be connected locally (analog, BRI, DCP, etc.) or remotely as Off-Premises stations. AWOH extensions count towards the maximum parties in a call. Trunks cannot be directly added to a call via this feature. Group extensions (e.g., hunt groups, PCOLs, TEGs, etc.) may not be added.

participationType [optional] Specifies the type of participation the added device has in the resulting call. Possible values are:

- PT_ACTIVE - the added device actively participates in the resulting conferenced call. As a result, the flow direction of the deviceToBeJoin's connection will be Transmit and Receive. Thus the added device can listen and talk.
- PT_SILENT - the added device can listen but cannot actively participate (cannot talk) in the resulting conferenced call. As a result, the flow direction of the deviceToBeJoin's connection will be Receive only. Thus the other parties on the call will be unaware that the added device has joined the call (no display updates). This option is useful for applications that may desire to silently conference in devices (e.g., service observing).
- If a party is Single Step Conferenced in with PT_SILENT, holds the call, and then conferences in another party, the PT_SILENT status of the original party is negated (which means the original party would then be heard by all other parties).

alertDestination [optional - partially supported] Specifies whether or not the deviceToBeJoin is to be alerted.

- TRUE - deviceToBeJoin will be alerted (with Delivered event) before it joins the call.

  Note: The "TRUE" option is not supported in the current release. If it is specified, the service request will fail with VALUE_OUT_OF_RANGE.

- FALSE - deviceToBeJoin will connect to the existing call without the device being alerted (no Delivered event). Only the "FALSE" option is supported in the current release.

Ack Parameter:

noData	None for this service.

Ack Private Parameters:

newCall	[mandatory] A connectionID specifies the callID and the deviceID of the joining device. The callID is the same callID as specified in the service request; that is, the callID of the resulting call is not changed.
connList	[optional - supported] Specifies the devices on the resulting newCall. This includes a count of the number of devices in the conferenced call and a list of connectionIDs and deviceIDs that define each connection in the call. If a device is on-PBX, the extension is specified. The extension consists of station or group extensions. Group extensions are provided when the conference is to a group and the conference completes before the call is answered by one of the group members (TEG, PCOL, hunt group, or VDN extension). It may contain alerting or bridged extensions. The static deviceID of a queued endpoint is set to the split extension of the queue. [optional - partially supported] Specifies whether or not the deviceToBeJoin is to be alerted. If a party is off-PBX, then its static device identified or its previously assigned trunk identifier is specified.
ucid	[optional - supported] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 .

- VALUE_OUT_OF_RANGE (3) (CS0/100) A not supported option is specified or some out-of-range value is specified in a parameter.
- OBJECT_NOT_KNOWN (4) (CS0/96) Mandatory parameter is missing.
- INVALID_CALLED_DEVICE (6) (CS0/28) The deviceToBeJoin is not a valid station or an AWOH extension, or an invalid callID is specified
- INVALID_CALLING_DEVICE (CS3/27) The deviceToBeJoin is on-hook when Single Step Conference is initiated. The deviceToBeJoin should be in off-hook/autoanswer condition.
- PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE (8) (CS3/43) The class of restriction on deviceToBeJoin is violated.
- INVALID_CSTA_DEVICE_IDENTIFIER (12) (CS0/28) The deviceToBeJoin is not a valid identifier.
- INVALID_FEATURE (15) (CS3/63) This feature is not supported on the switch. The switch software is prior to Release 6.
- INVALID_OBJECT_TYPE (18) (CS0/58) Call has conference restriction due to any of the data- related features (e.g., data restriction, privacy, manual exclusion, etc.).
- GENERIC_STATE_INCOMPATIBILITY (21) (CS0/18) The deviceToBeJoin cannot be forced off-hook and it did not go off-hook within 5 seconds.
- INVALID_OBJECT_STATE (22) (CS0/98) The request is made with option PT_ACTIVE while the call is in vector processing.
- RESOURCE_BUSY (33) (CS0/17) The deviceToBeJoin is busy or not in idle state.
- CONFERENCE_MEMBER_LIMIT_EXCEEDED (38) (CS3/42) The maximum allowed number of parties on the call has been reached.

Detailed Information:

- Bridged Call Appearance - A principal station with bridged call appearance can be single step conferenced into a call. Stations with bridged call appearance to the principal have the same bridged call appearance behavior, that is, if monitored, the station will receive Established And Conferenced Events when it joins the call. The station will not receive a Delivered Event.
- Call and Device Monitoring Event Sequences - A successful SingleStepConferenceCall request will generate an Established Event followed by a Conferenced Event for call monitoring and the monitoring of all devices that are involved in the newCall. The Established Event reports the connection state change of the DeviceToBeJoin and the Conferenced Event reports the result of the SingleStepConferenceCall request. All call-associated information (e.g., original calling and called device, UUI, collected digits, etc.) is reported in the Conferenced Event and Established Event. In both events, the cause value is EC_ACTIVE_MONITOR, if PT_ACTIVE was specified in the SingleStepConferenceCall request and EC_SILENT_MONITOR, if PT_SILENT was specified. The confController and addParty parameters in the Conferenced Event have the same device as specified by DeviceToBeJoin.

  The single step conference call event sequences are similar to the two-step conference call event sequences with one exception. Since the added party is alerted in the two-step conference call, a Delivered Event is generated. In a single-step conference call scenario, however; the deviceToBeJoin is added onto the call without alerting. Therefore, no Delivered Event is generated.
- Call State - The call into which a station is to be conferenced with Single Step Conference Service may be in any state, except the following situation. If the call is in vector processing and the PT_ACTIVE option is specified in the request, the request will be denied with INVALID_OBJECT_STATE. This will avoid interactions with vector steps such as "collect" when a party joins the call and is able to talk. If the PT_SILENT is specified, the request will be accepted.
- Dropping Recording Device - If single-step conference is used to add a recording device into a call, the application has the responsibility of dropping the recording device and/or call when appropriate. The DEFINITY switch cannot distinguish between recording devices and real stations, so if a recording device is left in the call with one other party, the DEFINITY switch will leave that call up forever, until one of those parties drops.
- Drop Button and Last Added Party - A party added by Single Step Conference Service will never be considered as "last added party" on the call. Thus, parties added through Single Step Conference Service would not be able to be dropped by using the Drop button.
- Feature Availability - The Single Step Conference Service is only available on the DEFINITY switch with Release 6 and later software. Software prior to R6 will deny the service request and return INVALID_FEATURE.
- Primary Old Call in Conferenced Event - Since the activeCall and the newCall parameters contain the same callID, there is no meaningful primaryOldCall in the Conferenced Event. The callID in primaryOldCall will have the value 0 and the deviceID will have the value "0" with type DYNAMIC.

- Remote Agent Trunk to Trunk Conference/Transfer - In this type application, an incoming call with an external caller is routed to a remote agent. The remote agent wants to transfer the call to another agent (also remote). Upon the agent's transfer request at the desktop, an application may use Single Step Conference Service to join a local device into this trunk-to-trunk call. This local device need not be a physical station; it may be a station AWOH. Having added the local station into the call, the application can hold the call and make a call to the new agent, and then transfer the call. The caller is now connected to the second remote agent, and the local station (physical or AWOH) that was used to accomplish the transfer is no longer on the call.
- State of Added Station - A station to be conferenced into a call must be idle. A station is considered idle when it has an idle call appearance for call origination. If a station is off-hook idle when the Single Step Conference Service is received, the station is immediately conferenced in. If a station is on-hook idle and it may be forced off-hook, it will be forced off-hook and immediately conferenced in. If a station is on-hook idle and it may not be forced off-hook, the switch will wait 5 seconds for the user to go off-hook. If the user does not go off-hook within 5 seconds, then a negative acknowledgment with GENERIC_STATE_INCOMPATIBILITY is sent.
- Security - As long as it is allowed by switch administration, an application can add a party onto a call with Single Step Conference Call Service without any audible signal or visual display to the existing parties on the call. If security is a concern, proper switch administration must be performed.

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t         cstaEscapeService (
    ACSHandle_t       acsHandle,
    InvokeID_t        invokeID,
    PrivateData_t     *privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
    ACSHandle_t       acsHandle;
    EventClass_t      eventClass;// CSTACONFIRMATION
    EventType_t       eventType; // CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t      eventHeader;
    union
    {
        struct
        {
            InvokeID_t   invokeID;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSingleStepConferenceCall() - Service Request Private Data
                                 // Setup Function

RetCode_t    attSingleStepConferenceCall(
     ATTPrivateData_t        *privateData,
     ConnectionID_t          *activeCall,       // mandatory
     DeviceID_t              *deviceToBeJoin,   // mandatory
     ATTParticipationType_t  participation,
     Boolean                 alertDestination);

typedef struct ATTPrivateData_t {
     char     vendor[32];
     ushort   length;
     char     data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTSingleStepConferenceCallConfEvent - Private Data Service Response

typedef struct
{
     ATTEventType_t     eventType;
                             // ATT_SINGLE_STEP_CONFERENCE_CALL_CONF
     union
     {
          ATTSingleStepConferenceCallConfEvent_t ssconference;
     }u;
     char heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct Connection_t {
     ConnectionID_t  party;
     DeviceID_t      staticDevice;  // NULL for not present
} Connection_t;

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

typedef struct ATTSingleStepConferenceCallConfEvent_t {
     ConnectionID_t     newCall;
     ConnectionList_t   connList;
     ATTUCID_t          ucid;
} ATTSingleStepConferenceCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Single Step Transfer Call (Private Data Version 8 and later)

Summary

- Direction: Client to Switch
- Function: *cstaEscapeService()*
- Confirmation Event: *CSTAEscapeServiceConfEvent*
- Private Data Function: *attSingleStepTransferCall()*
- Private Data Confirmation Event: *ATTSingleStepTransferCallConfEvent*
- Service Parameters: *noData*
- Private Parameters: *activeCall, transferredTo*
- Ack Parameters: *noData*
- Ack Private Parameters: *transferredCall*
- Nak Parameter: *universalFailure*

Functional Description:

The Single Step Transfer Call service transfers an existing connection to another device. This transfer is performed in a single step. This means that the device transferring the call does not have to place the existing call on hold before issuing the Single Step Transfer Call service.

The connection being transferred may be in the Alerting, Connected, Held, or Queued state.

Service Parameters:

noData	None for this service.

Private Parameters:

activeCall	[mandatory] A pointer to the connection identifier of the active call which is to be transferred.
transferredTo	[mandatory] A pointer to the destination address to which the call will be transferred.

Ack Parameter:

noData None for this service.

Ack Private Parameters:

transferredCall [mandatory] Specifies the connection ID for the destination of the transferred call.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 .

- OBJECT_NOT_KNOWN (4) (CS0/96) The activeCall does not contain a call ID, or transferredTo is not set.
- INVALID_CALLED_DEVICE (6) (CS0/28) The transferredTo device is not a valid transfer destination. It might be blocked by the transferring device's Class of Restriction (COR).
- INVALID_CSTA_DEVICE_IDENTIFIER (12) (CS0/28) The transferring device is not a valid extension.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13)
  - The call id in activeCall is not an active call id.
  - The call id in activeCall is not present at the transferring device.
- GENERIC_STATE_INCOMPATIBILITY (21) The active call is alerting.
- INVALID_OBJECT_STATE (22) (CS0/98) The active call is alerting at the transferring device.
- RESOURCE_BUSY (33) (CS0/17)
  - The transferring device does not have an available call appearance or the call appearance is restricted from originating a new call.
  - The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, etc.) to the same device.

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t         cstaEscapeService (
     ACSHandle_t       acsHandle,
     InvokeID_t        invokeID,
     PrivateData_t     *privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
     ACSHandle_t       acsHandle;
     EventClass_t      eventClass;// CSTACONFIRMATION
     EventType_t       eventType; // CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
     ACSEventHeader_t      eventHeader;
     union
     {
          struct
          {
               InvokeID_t   invokeID;
          } cstaConfirmation;
     } event;
     char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Data Version 8 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSingleStepTransferCall() - Service Request Private Data
                                 // Setup Function

RetCode_t    attSingleStepTransferCall(
     ATTPrivateData_t        *privateData,
     ConnectionID_t          *activeCall,       // mandatory
     DeviceID_t              *transferredTo)    // mandatory

typedef struct ATTPrivateData_t {
     char      vendor[32];
     ushort    length;
     char      data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTSingleStepTransferCallConfEvent - Private Data Service Response

typedef struct
{
     ATTEventType_t     eventType;
                             // ATT_SINGLE_STEP_TRANSFER_CALL_CONF
     union
     {
          ATTSingleStepTransferCallConfEvent_t ssTransferCallConf;
     }u;
     char heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTSingleStepTransferCallConfEvent_t {
     ConnectionID_t     transferredCall;
} ATTSingleStepTransferCallConfEvent_t;
```

Transfer Call Service

Summary

- Direction: Client to Switch
- Function: *cstaTransferCall ()*
- Confirmation Event: *CSTATransferCallConfEvent*
- Private Data Confirmation Event: *ATTTransferCallConfEvent*
- Service Parameters: *heldCall, activeCall*
- Ack Parameters: *newCall, connList*
- Ack Private Parameters: *ucid*
- Nak Parameter: *universalFailure*

Functional Description:

This service provides the transfer of an existing held call (heldCall) and another active or proceeding call (alerting, queued, held, or connected) (activeCall) at a device provided that heldCall and activeCall are not both in the alerting state at the controlling device. The Transfer Call Service merges two calls with connections at a single common device into one call. Also, both of the connections to the common device become Null and their connectionIDs are released. A connectionID that specifies the resulting new connection for the transferred call is provided.

Service Parameters:

heldCall	[mandatory] Must be a valid connection identifier for the call that is on hold at the controlling device and is to be transferred to the activeCall. The deviceID in heldCall must contain the station extension of the controlling device.
activeCall	[mandatory] Must be a valid connection identifier of an active or proceeding call at the controlling device to which the heldCall is to be transferred. The deviceID in activeCall must contain the station extension of the controlling device.

Ack Parameters:

newCall [mandatory - partially supported] A connection identifier that specifies the resulting new call identifier for the transferred call.

connList [optional - supported] Specifies the devices on the resulting new call. This includes a count of the number of devices in the new call and a list of up to six connectionIDs and up to six deviceIDs that define each connection in the call.

- If a device is on-PBX, the extension is specified. The extension consists of station or group of extensions. Group extensions are provided when the conference is to a group and the conference completes before the call is answered by one of the group members (TEG, PCOL, hunt group, or VDN extension). It may contain alerting extensions.
- The static deviceID of a queued endpoint is set to the split extension of the queue.
- If a party is off-PBX, then its static device identifier or its previously assigned trunk identifier is specified.

Ack Private Parameters:

ucid [optional] Specifies the Universal Call ID (UCID) of newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in heldCall or activeCall.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) The controlling deviceID in activeCall or heldCall has not been specified correctly.
- GENERIC_STATE_INCOMPATIBILITY (21) Both calls are alerting. Both calls are being service-observed. An active call is in a vector-processing stage.
- INVALID_OBJECT_STATE (22) The connections specified in the request are not in the valid states for the operation to take place. For example, it does not have one active call and one held call as required.
- INVALID_CONNECTION_ID_FOR_ACTIVE_CALL (23) The callID in activeCall or heldCall has not been specified correctly.
- RESOURCE_BUSY (33) The switch is busy with another CSTA request. This can happen when two TSAPI Services are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Transfer Call, etc.) to the same device.
- CONFERENCE_MEMBER_LIMIT_EXCEEDED (38) The request attempted to add a seventh party to an existing six-party conference call.
- MISTYPED_ARGUMENT_REJECTION (74) DYNAMIC_ID is specified in heldCall or activeCall.

Detailed Information:

- Analog Stations - Transfer Call Service will only be allowed if one call is held and the second is active (talking). Calls on hard-held or alerting cannot be affected by a Transfer Call Service.
  - An analog station will support Transfer Call Service even if the "switch-hook flash" field on the Communication Manager system administered form is set to "no." A "no" in this field disables the switch- hook flash function, meaning that a user cannot conference, hold, or transfer a call from his/her phone set, and cannot have the call waiting feature administered on the phone set.
  - Bridged Call Appearance - Transfer Call Service is not permitted on parties in the bridged state and may also be more restrictive if the principal of the bridge has an analog station or the exclusion option is in effect from a station associated with the bridge or PCOL.

- Trunk to Trunk Transfer - Existing rules for trunk-to-trunk transfer from a station user will remain unchanged for application monitored calls. In such case, transfer requested via Transfer Call Service will be denied. When this feature is enabled, application monitored calls transferred from trunk to trunk will be allowed, but there will be no further event reports (except for the Network Reached, Established, or Connection Cleared Event Reports sent to the application).

Syntax

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

// cstaTransferCall() - Service Request

RetCode_t         cstaTransferCall (
    ACSHandle_t       acsHandle,
    InvokeID_t        invokeID,
    ConnectionID_t    *heldCall,        // devIDType= STATIC_ID
    ConnectionID_t    *activeCall,      // devIDType= STATIC_ID
    PrivateData_t     *privateData);

// CSTATransferCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t       acsHandle;
    EventClass_t      eventClass;// CSTACONFIRMATION
    EventType_t       eventType; // CSTA_TRANSFER_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t      eventHeader;
    union
    {
        struct
        {
            InvokeID_t   invokeID;
            union
            {
            CSTATransferCallConfEvent_t        transferCall;
            } u;
        } cstaConfirmation;
    } event;
    char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct Connection_t {
ConnectionID_t    party;
DeviceID_t        staticDevice; // NULL for not present
} Connection_t;

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

typedef struct {
    ConnectionID_t        newCall;
    ConnectionListID_t    connList;
} CSTATransferCallConfEvent_t;
```

Private Data Version 5 Syntax

```
// ATTTransferCallConfEvent -  Service Response Private
// Data (supported by private data version 5 and later only)

typedef struct
{
     ATTEventType_t     eventType;// ATT_TRANSFER_CALL_CONF
     union
     {
     ATTTransferCallConfEvent_t            transferCall;
     }u;
} ATTEvent_t;

typedef struct ATTTransferCallConfEvent_t
{
     ATTUCID_t     ucid;
} ATTTransferCallConfEvent_t;

typedef char ATTUCID_t[64];
```

Chapter 7: Set Feature Service Group

Set Feature Service Group describes services that allow a client application to set switch-controlled features or values on a G3 device. The following sections describes the Set Features services that Application Enablement Services (AE Services) supports.

Set Advice of Charge Service (Private Data Version 5 and Later)

Summary

- Direction: Client to Switch
- Function: cstaEscapeService()
- Confirmation Event: CSTAEscapeConfEvent
- Private Data Function: attSetAdviceOfCharge()
- Service Parameters: noData
- Private Parameters: featureFlag
- Ack Parameters: noData
- Ack Private Parameters: noData
- Nak Parameter: universalFailure

Functional Description

DEFINITY ECS Release 5 and later software supports the Charge Advice Event feature. To receive Charge Advice Events, an application must first turn the Charge Advice Event feature on using the Set Advice of Charge Service (Private Data V5).

If the Charge Advice Event feature is turned on, a trunk group monitored by a cstaMonitorDevice(), a station monitored by a cstaMonitorDevice(), or a call monitored by a cstaMonitorCall or cstaMonitorCallsViaDevice will receive Charge Advice Events. However, this will not occur if the Charge Advice Event is filtered out by the privateFilter in the monitor request and its confirmation event.

This service enables the DEFINITY to support the collection of charging units over ISDN Primary Rate Interfaces. See Detailed Information: on page 339 for more information about this feature. See also, Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

Service Parameters:

noData	None for this service.

Private Parameters:

featureFlag	[mandatory] Specify the flag for turning the feature on or off. A value of TRUE will turn the feature on and a value of FALSE will turn the feature off. If the feature is already turned on, subsequent requests to turn the feature on again will receive positive acknowledgements. If the feature is turned off, subsequent requests to turn the feature off again will receive positive acknowledgements.

Ack Parameters:

noData	None for this service.

Ack Private Parameters:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_FEATURE (15) The Set Advice of Charge Service (Private Data V5) is not supported by the switch. VALUE_OUT_OF_RANGE (3) The featureFlag contains an invalid value.

Detailed Information:

- The result of a successful Set Advice of Charge Service (Private Data V5 and later) request applies to an ACS Stream. This means that any program using the same acsHandle will be affected. An application must use the private filter to filter out Advice Of Charge Events, if these events are not useful to the application.
- If this feature is heavily used, it will reduce the maximum Busy Hour Call Completions (BHCC) of the DEFINITY.
- If more than 100 calls are in a call clearing state waiting for charging information, the oldest record will not receive final charge information. In this case a value of 0 and a cause value of EC_NETWORK_CONGESTION will be reported in the Advice of Charge Event.
- For information about administering the switch for using Advice of Charge, see "Administering Advice of Charge for ASAI Charging Event," in Appendix A of the *Avaya MultiVantage Application Enablement Services ASAI Technical Reference*, 03-300549.

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
            ACSHandle_t             acsHandle,
            InvokeID_t              invokeID,
            PrivateData_t*privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTACONFIRMATION
            EventType_t eventType;  // CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
            ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
                                    InvokeID_t  invokeID;
                        } cstaConfirmation;
            } event;
            char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSetAdviceOfCharge() - Service Request Private Data Setup Function

RetCode_t   attAdviceOfCharge(
            ATTPrivateData          *privateData,
Boolean                             featureFlag);

typedef struct ATTPrivateData_t {
            char        vendor[32];
            ushort      length;
            char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;
```

Set Agent State Service

Summary

- Direction: Client to Switch
- Function: cstaSetAgentState()
- Confirmation Event: CSTASetAgentStateConfEvent
- Private Data Function: attV6SetAgentState (private data versions 6 and 7), attSetAgentStateExt (private data version 5), attSetAgentState (private data versions 2-4)
- Service Parameters: device, agentMode, agentID, agentGroup, agentPassword
- Private Parameters: workMode, reasonCode, enablePending
- Ack Parameters: noData
- Ack Private Parameters: isPending
- Nak Parameter: universalFailure

Functional Description:

This service allows a client to log an ACD agent into or out of a G3 ACD Split and to specify a change of work mode for a G3 ACD agent.

Service Parameters:

device [mandatory] Specifies the agent extension. This must be a valid on-PBX station extension for an ACD agent.

agentMode [mandatory - partially supported] Specifies log in or log out for an Agent into or out of an ACD split, or a change of work mode for an Agent logged into an ACD split:

- AM_LOG_IN - Log in the Agent. This does not imply that the Agent is ready to accept calls. The initial mode for the ACD agent can be set via the workMode private parameter (see the private parameter workMode). If the workMode private parameter is not supplied, the initial work mode for the ACD agent will be set to the G3 specific "Auxiliary-Work Mode".
- AM_LOG_OUT - Log an Agent out of a specific ACD split. The Agent will be unable to accept additional calls for the ACD split.
- AM_NOT_READY - Change the work mode for an Agent logged into an ACD split to "Not Ready" (equivalent to G3 "Auxiliary-Work Mode"), indicating that the Agent is occupied with some task other than serving a call.
- AM_READY - Change the work mode for Agent logged into an ACD split to "Ready". The Agent in the Ready state is ready to accept calls or is currently busy with an ACD call. The workMode private parameter may be used to set the ACD agent work mode to the ATT specific "Auto-In-Work Mode" or "Manual-In-Work Mode". If the workMode private parameter is not supplied, the ACD agent work mode will be set to the ATT specific "Auto-In-Work Mode".
- AM_WORK_NOT_READY - Change the work mode for an Agent logged into an ACD split to "Work Not Ready" (equivalent to Communication Manager "After-Call-Work Mode"). The Agent in the Work Not Ready state is occupied with the task of serving a call after the call has disconnected, and the Agent is not ready to accept additional calls for the ACD split.
- AM_WORK_READY - A change to "Work Ready" is not currently supported for Communication Manager.

agentID [optional] Specifies the Agent login identifier for the ACD agent. This parameter is mandatory when the agentMode parameter is AM_LOG_IN; otherwise it is ignored. An agentID containing a Logical Agent's login Identifier can be used to log in a Logical Agent (Expert Agent Selection [EAS]) when paired with the agentPassword.

agentGroup [mandatory] Specifies the ACD agent split to use to log in, log out, or change the agent work mode to "Not Ready", "Ready" or "Work Not Ready". In an Expert Agent Selection (EAS) environment, the agentGroup parameter must contain the skill group extension.

agentPassword [optional - partially supported] Specifies a password that allows an ACD agent to log into an ACD Split. This service parameter is only used if agentMode is set to AM_LOG_IN; otherwise it is ignored. The agentPassword can be used to log in a Logical Agent (with EAS) when included with the Logical Agent's login Identifier, the agentID.

Private Parameters:

workMode [optional] Specifies the work mode for the agent as Auxiliary- Work Mode (WM_AUX_WORK), After-Call-Work Mode (WM_AFT_CALL), Auto-In Mode (WM_AUTO_IN), or Manual- In-Work Mode (WM_MANUAL_IN) based on the agentMode service parameter as follows:

- AM_LOG_IN - The workMode private parameter specifies the initial work mode for the ACD agent. Valid values include "Auxiliary-Work Mode" (Default), "After-Call-Work Mode", "Auto-In Mode", or "Manual-In Mode".
- AM_LOG_OUT - The workMode is ignored.
- AM_NOT_READY - The workMode is ignored.
- AM_READY - The workMode private parameter specifies the work mode for the ACD agent. Valid values include "Auto-In-Work Mode" (Default), or "Manual-In-Work Mode".
- AM_WORK_NOT_READY - The workMode is ignored.
- AM_WORK_READY - The workMode is ignored.

reasonCode [optional] Specifies the reason for change of work mode to WM_AUX_WORK or the logged-out (AM_LOG_OUT) state.

For private data version 7 valid reason codes range from 0 to 99. A value of 0 indicates that the reason code is not available. The meaning of the codes 1 through 99 is defined by the application. This range of reason codes is supported by private data version 7 only.

Private data versions 6 and 5 support single digit reason codes 1 through 9. A value of 0 indicates that the reason code is not available. The meaning of the code (1-9) is defined by the application.

Private data version 4 and earlier do not support reason codes.

enablePending [optional] Specifies whether the requested change can be made pending.

A value of TRUE will enable the pending feature. If the agent is busy on a call when an attempt is made to change the agentMode to AM_NOT_READY or AM_WORK_NOT_READY, and enablePending is set to TRUE, the change will be made *pending* and will take effect as soon as the agent clears the call. The request will be acknowledged (Ack).

If enablePending is not set to TRUE and the agent is busy on a call, the requested change will not be made *pending* and the request will not be acknowledged (Nak).

Note: Subsequent requests may override a pending change and only the most recent pending change will take effect when the call is cleared. The enablePending parameter applies to the reasonCode when the request is to change the agentMode to AM_NOT_READY.

This parameter is supported by private data version 6 and later only.

Ack Parameters:

noData None for this service.

Ack Private Parameters:

isPending	[optional] If isPending is set to TRUE, the requested change in workmode is pending. Otherwise, the requested change took effect immediately.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 • GENERIC_UNSPECIFIED (0) An attempt to log out an ACD agent who is already logged out, an attempt to log in an ACD agent to a split of which they are not a member, or an attempt to log in an ACD agent with an incorrect password. • GENERIC_OPERATION (1) An attempt to log in an ACD agent that is already logged in. • VALUE_OUT_OF_RANGE (3) The workMode private parameter is not valid for the agentMode (see Table 13). • The reason code is out of the acceptable range (1- 9). (CS0/100). • OBJECT_NOT_KNOWN (4) Did not specify a valid on-PBX station for the ACD agent in device, the agentGroup or device parameters were NULL, or the agentID parameter was NULL when agentMode was set to AM_LOG_IN. • INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device. • INVALID_FEATURE (15) The feature is not available for the agentGroup, or the enablePending feature is not available for the administered switch version. • INVALID_OBJECT_TYPE (18) (CS3/80) The reason code is specified, but the workMode is not WM_AUX_WORK, or agentMode is not AM_LOG_OUT. • GENERIC_STATE_INCOMPATIBILITY (21) A work mode change was requested for a non-ACD agent, or the Agent station is maintenance busy or out of service. • INVALID_OBJECT_STATE (22) The Agent is already logged into another split. • GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31) The request cannot be completed due to lack of available switch resources. • RESOURCE_BUSY (33) Attempt to log in an ACD agent that is currently on a call.

Detailed Information:

- A request to log in an ACD agent (agentMode is AM_LOG_IN) that does not include the private parameter workMode, will set the initial Agent work state to Auxiliary-Work Mode (Not Ready).
- The AM_WORK_READY agentMode is not supported by Communication Manager.
- The agentPassword service parameter applies only for requests to log in an ACD agent (agentMode is AM_LOG_IN). In all other cases, it is ignored. The agentPassword can be used to log in a Logical Agent (with Expert Agent Selection [EAS]) when included with the Logical Agent's login Identifier, the agentID.
- Valid combinations of the agentMode service parameter and the workMode, reasonCode, and enablePending private parameters are shown in Table 13.

Table 13: agentMode Service Parameter and Associated Private Parameters

agentMode	workMode	Reason code	enablePending
AM_LOG_IN	WM_AUX_WORK (Default)	1-99	NA
	WM_AFTCAL_WK	ignored	
	WM_AUTO_IN	ignored	
	WM_MANUAL_IN	ignored	
AM_LOG_OUT	WM_AUX_WORK (Default)	1-99	NA
	WM_AFTCAL_WK	ignored	
	WM_AUTO_IN	ignored	
	WM_MANUAL_IN	ignored	
AM_NOT_READY	NA	1-99	TRUE/FALSE
AM_READY	WM_AUTO_IN (Default) WM_MANUAL_IN	NA	NA
AM_WORK_NOT_READY	NA	NA	TRUE/FALSE

- AttSetAgentStateExt() and attSetAgentState() do not accept the enablePending parameter. These functions will never cause the requested work mode change to be made pending, even if the switch is G3V8 or later.
- Subsequent pending work mode requests replace earlier requests.

Syntax

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

//  cstaSetAgentState() - Service Request

RetCode_t   cstaSetAgentState (
             ACSHandle_t  acsHandle,
             InvokeID_t  invokeID,
             DeviceID_t  *device,
             AgentMode_t  agentMode,
             AgentID_t  *agentID,
             AgentGroup_t  *agentGroup,
             AgentPassword_t  *agentPassword,
             PrivateData_t  *privateData);

typedef char              DeviceID_t[64];

typedef enum AgentMode_t {
             AM_LOG_IN                = 0,
             AM_LOG_OUT               = 1,
             AM_NOT_READY             = 2,
             AM_READY                 = 3,
             AM_WORK_NOT_READY        = 4,
             AM_WORK_READY            = 5
} AgentMode_t;

typedef char              AgentID_t[32];

typedef DeviceID_t        AgentGroup_t;

typedef char              AgentPassword_t[32];

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

//  CSTASetAgentStateConfEvent - Service Response

typedef struct
{
             ACSHandle_t              acsHandle;
             EventClass_t  eventClass;  // CSTACONFIRMATION
             EventType_t   eventType;   // CSTA_SET_AGENT_STATE_CONF
} ACSEventHeader_t;

typedef struct CSTASetAgentStateConfEvent_t {
             Nulltype     null;
```

Syntax (Continued)

```
} CSTASetAgentStateConfEvent_t;

typedef struct
{
            ACSEventHeader_t  eventHeader;

            union
            {
                        struct
                        {
                                    InvokeID_t  invokeID;
                                    union
                                    {
                                                CSTASetAgentStateConfEvent_t
setAgentState;
            } u;
            } cstaConfirmation;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attV6SetAgentState() - Service Request Private Data Setup Function

RetCode_t    attV6SetAgentState(
             ATTPrivateData_t  *attPrivateData,
             ATTWorkMode_t workMode,                  // Work Modes
             long          reasonCode,                // single digit 1-9
             Boolean       enablePending);            // TRUE = enabled

typedef struct ATTPrivateData_t
{
             char          vendor[32];
             ushort        length;
             char          data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTWorkMode_t
{
             WM_AUX_WORK   = 1,          // Same As C_AUX_WORK
             WM_AFTCAL_WK  = 2,          // Same as C_AFTCAL_WK
             WM_AUTO_IN    = 3,          // Same as C_AUTO_IN
             WM_MANUAL_IN  = 4           // Same as C_MANUAL_IN
} ATTWorkMode_t;

// ATTSetAgentStateConfEvent - Confirmation Event Private Data

typedef struct
{
             ATTEventType  eventType;    // ATT_SET_AGENT_STATE_CONF
             union
             {
             ATTSetAgentStateConfEvent_t setAgentState;

             }u;
             char          heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTSetAgentStateConfEvent_t {
             Boolean       isPending; // TRUE if the request is pending
} ATTSetAgentStateConfEvent_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSetAgentState() - Service Request Private Data Setup Function

RetCode_t    attSetAgentState(
             ATTPrivateData_t  *attPrivateData,
             ATTWorkMode_t workMode,                  // Work Modes
             long          reasonCode);               // single digit 1-9

typedef struct ATTPrivateData_t
{
             char          vendor[32];
             ushort        length;
             char          data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTWorkMode_t
{
             WM_AUX_WORK   = 1,          // Same As C_AUX_WORK
             WM_AFTCAL_WK  = 2,          // Same as C_AFTCAL_WK
             WM_AUTO_IN    = 3,          // Same as C_AUTO_IN
             WM_MANUAL_IN  = 4           // Same as C_MANUAL_IN
} ATTWorkMode_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSetAgentState() - Service Request Private Data Setup Function

RetCode_t    attSetAgentState(
             ATTPrivateData_t  *attPrivateData,
             ATTWorkMode_t workMode);                // Work Modes

typedef struct ATTPrivateData_t
{
             char         vendor[32];
             ushort       length;
             char         data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTWorkMode_t
{
             WM_AUX_WORK   = 1,          // Same As C_AUX_WORK
             WM_AFTCAL_WK  = 2,          // Same as C_AFTCAL_WK
             WM_AUTO_IN    = 3,          // Same as C_AUTO_IN
             WM_MANUAL_IN  = 4           // Same as C_MANUAL_IN
} ATTWorkMode_t;
```

Set Billing Rate Service (Private Data Version 5 and Later)

Summary

- Direction: Client to Switch
- Function: cstaEscapeService()
- Confirmation Event: CSTAEscapeConfEvent
- Private Data Function: attSetBillingRate()
- Service Parameters: noData
- Private Parameters: call, billType, billRate
- Ack Parameters: noData
- Nak Parameter: universalFailure

Functional Description:

This service supports the AT&T MultiQuest® 900 Vari-A-Bill Service to change the rate for an incoming 900-type call. The client application can request this service at any time after the call has been answered and before the call is cleared.

Service Parameters:

noData	None for this service.

Private Parameters:

call	[mandatory] Specifies the call to which the billing rate is to be applied. This is a connection identifier, but only the callID is used. The deviceID of call is ignored.
billType	[mandatory] Specifies the rate treatment for the call and can be one of the following: ● BT_NEW_RATE ● BT_FLAT_RATE (time independent) ● BT_PREMIUM_CHARGE (i.e., a flat charge in addition to the existing rate) ● BT_PREMIUM_CREDIT (i.e., a flat negative charge in addition to the existing rate) ● BT_FREE_CALL
billRate	[mandatory] Specifies the rate according to the treatment indicated by billType. If FREE_CALL is specified, billRate is ignored. This is a floating point number. The rate should not be less than $0 and a maximum is set for each 900-number as part of the provisioning process (in the 4E switch)

Ack Parameters:

noData	None for this service.

Nak Parameter

:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.
- INVALID_CSTA_CONNECTION_IDENTIFIER (13) An invalid connection identifier has been specified in call.
- VALUE_OUT_OF_RANGE (3) (CS0/96) Invalid value is specified in the request.
- INVALID_OBJECT_STATE (22) (CS0/98) The request is attempted before the call is answered.
- RESOURCE_BUSY (33) (CS0/47) The switch limit for unconfirmed requests has been reached.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) (CS0/29) The user has not subscribed to the Set Billing Rate Service (Private Data V5 and later) feature.

Detailed Information:

None

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
            ACSHandle_t             acsHandle,
            InvokeID_t              invokeID,
            PrivateData_t           *privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_t eventClass;// CSTACONFIRMATION
            EventType_t eventType;  // CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
            ACSEventHeader_t eventHeader;
            union
            {
            struct
                        {
                        InvokeID_t  invokeID;
                        } cstaConfirmation;
            } event;
            char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSetBillingRate() - Service Request Private Data Setup Function

RetCode_t   attSetBillingRate(
            ATTPrivateData          *privateData,
            ConnectionID_t          *call,
            ATTBillType_t           billType,
            float                   billRate);

typedef struct ATTPrivateData_t {
            char                    vendor[32];
            ushort                  length;
            char                    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTBillType_t {
            BT_NEW_RATE             = 16,
            BT_FLAT_RATE            = 17,
            BT_PREMIUM_CHARGE       = 18,
            BT_PREMIUM_CREDIT       = 19,
            BT_FREE_CALL            = 24
} ATTBillType_t;
```

Set Do Not Disturb Feature Service

Summary

- Direction: Client to Switch
- Function: cstaSetDoNotDisturb()
- Confirmation Event: CSTASetDndConfEvent
- Service Parameters: device, doNotDisturb
- Ack Parameters: noData
- Nak Parameter: universalFailure

Functional Description:

This service turns on or off the G3 Send All Calls (SAC) feature for a user station.

Service Parameters:

device	[mandatory] Must be a valid on-PBX station extension that supports the SAC feature.
doNotDisturb	[mandatory] Specifies either "On" (TRUE) or "Off" (FALSE).

Ack Parameters:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 • INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device. • GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) This error is returned if the station does not have a coverage path.

Detailed Information:

- DCS - SAC feature may not be requested by this service for an off-PBX DCS extension.
- Logical Agents - SAC may not be requested by this service for logical agent login IDs. If a login ID is specified, the request will be denied (INVALID_CSTA_DEVICE_IDENTIFIER). SAC may be requested by this service on behalf of a logical agent's station extension. In an Expert Agent Selection (EAS) environment, if the call is made to a logical agent ID, the call coverage follows the path administered for the logical agent ID, and not the coverage path of the physical set from which the agent is logged in. SAC cannot be activated by a CSTA request for the logical agent ID.
- Send All Calls (SAC) - This G3 feature allows users to temporarily direct all incoming calls to coverage regardless of the assigned Call Coverage redirection criteria. Send All Calls also allows covering users to temporarily remove their voice terminals from the coverage path. SAC is used only in conjunction with the Call Coverage feature. Details of how SAC is used in conjunction with the Call Coverage are documented in the DEFINITY Communications System Generic 3 Feature Description, 555-230-201.

Syntax

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

//  cstaSetDoNotDisturb() - Service Request

RetCode_t   cstaSetDoNotDisturb (
            ACSHandle_t acsHandle,
            InvokeID_t  invokeID,
            DeviceID_t  *device,
            Boolean     doNotDisturb,// TRUE = On  or FALSE = Off
            PrivateData_t *privateData);

typedef char            DeviceID_t[64];
typedef char            Boolean;

//  CSTASetDndConfEvent - Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_t eventClass;            // CSTACONFIRMATION
            EventType_t eventType;              // CSTA_SET_DND_CONF
} ACSEventHeader_t;

typedef struct CSTASetDndConfEvent_t {
    Nulltype        null;
} CSTASetDndConfEvent_t;

typedef struct
{
            ACSEventHeader_teventHeader;
            union
            {
            struct
                        {
                        InvokeID_t  invokeID;
                        union
                        {
                        CSTASetDndConfEvent_tsetDnd;
                        } u;
                        } cstaConfirmation;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Set Forwarding Feature Service

Summary

- Direction: Client to Switch
- Function: cstaSetForwarding()
- Confirmation Event: CSTASetFwdConfEvent
- Service Parameters: device, forwardingType, forwardingOn, forwardingDN
- Ack Parameters: noData
- Nak Parameter: universalFailure

Functional Description:

The Set Forwarding Service sets the G3 Call Forwarding feature on or off for a user station. G3 CSTA supports the Immediate type of forwarding only.

Service Parameters:

device	[mandatory] Specifies the station on which the Call Forwarding feature is to be set. It must be a valid on-PBX station extension that supports the Call Forwarding feature.
forwardingType	[mandatory - partial] Specifies the type of forwarding to set or clear. Only FWD_IMMEDIATE is supported. Any other types will be denied.
forwardingOn	[mandatory] Specifies "On" (TRUE) or "Off" (FALSE).
forwardingDN	[mandatory] Specifies the station extension to which the calls are to be forwarded. It is mandatory if forwardingOn is set to on. It is ignored by Communication Manager if the forwardingOn is set to off.

Ack Parameters:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device or forwardingDN. GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) The user has not subscribed to the Call Forwarding feature.

Detailed Information:

- DCS - The Call Forwarding feature may not be activated by this service for an off-PBX DCS extension.
- Logical Agents - Call Forwarding may not be requested by this service for logical agent login IDs. If a login ID is specified as the forwardingDN, the request will be denied (INVALID_CSTA_DEVICE_IDENTIFIER). Call Forwarding may be requested on behalf of a logical agent's station extension.
- G3 Call Forwarding All Calls - This feature allows all calls to an extension number to be forwarded to a selected internal extension number, external (off-premises) number, the attendant group, or a specific attendant. It supports only the CSTA forwarding type "Immediate."
- Activation and Deactivation - Activation and deactivation from the station and a client application may be intermixed.

Syntax

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

//  cstaSetForwarding() - Service Request

RetCode_t   cstaSetForwarding (
            ACSHandle_t acsHandle,
            InvokeID_t  invokeID,
            DeviceID_t  *device,
            ForwardingType_t forwardingType,// must be FWD_IMMEDIATE
            Boolean     forwardingOn,// TRUE (on) or FALSE (off)
            DeviceID_t  *forwardingDestination,
            PrivateData_t*privateData);

typedef char            DeviceID_t[64];

typedef enum ForwardingType_t {
    FWD_IMMEDIATE       = 0,        // Only option supported
    FWD_BUSY            = 1,                    // Not supported
    FWD_NO_ANS          = 2,                    // Not supported
    FWD_BUSY_INT        = 3,                    // Not supported
    FWD_BUSY_EXT        = 4,                    // Not supported
    FWD_NO_ANS_INT      = 5,                    // Not supported
    FWD_NO_ANS_EXT      = 6                     // Not supported
} ForwardingType_t;

typedef char            Boolean;

//  CSTASetFwdConfEvent - Service Response

typedef struct
{
            ACSHandle_t             acsHandle;
            EventClass_t            eventClass; // CSTACONFIRMATION
            EventType_t             eventType;  // CSTA_SET_FWD_CONF
} ACSEventHeader_t;

typedef struct CSTASetFwdConfEvent_t {
    Nulltype        null;
} CSTASetFwdConfEvent_t;
```

Syntax (Continued)

```
typedef struct
{
            ACSEventHeader_t eventHeader;
            union
            {
            struct
            {
                        InvokeID_t  invokeID;
                        union
                        {
                        CSTASetFwdConfEvent_t setFwd;
                        } u;

                        } cstaConfirmation;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Set Message Waiting Indicator (MWI) Feature Service

Summary

- Direction: Client to Switch
- Function: cstaSetMsgWaitingInd()
- Confirmation Event: CSTASetMwiConfEvent
- Service Parameters: device, messages
- Ack Parameters: noData
- Nak Parameter: universalFailure

Functional Description:

This service sets the G3 Message Waiting Indicator (MWI) on or off for a user station.

Service Parameters:

device	[mandatory] Must be a valid on-PBX station extension that supports the MWI feature.
messages	[mandatory] Specifies either "On" (TRUE) or "Off" (FALSE).

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786. INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

- Adjunct Messages - When a client application has turned on a station’s MWI and the station user retrieves message using the station display, then the station display will show "You have adjunct messages."
- MWI Status Sync - To keep the MWI synchronized with other applications, a client application must use this service to update the MWI whenever the link between the switch and the PBX Driver comes up from a cold start. An application can query the MWI status through the CSTAQueryMsgWaitingInd Service.
- System Starts - System cold starts will cause the switch to lose the MWI status. Hot starts (PE interchange) and warm starts will not affect the MWI status.
- Voice (Synthesized) Message Retrieval - A recording, "Please call message center for more messages," will be used for the case when the MWI has been activated by the application through this service.

Syntax

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

//  cstaSetMsgWaitingInd() - Service Request

RetCode_t   cstaSetMsgWaitingInd (
            ACSHandle_t acsHandle,
            InvokeID_t  invokeID,
            DeviceID_t  *device,
            Boolean     messages,   // TRUE (on) or FALSE (off)
            PrivateData_t *privateData);

typedef char            DeviceID_t[64];
typedef char            Boolean;

//  CSTASetMwiConfEvent - Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_t eventClass;            // CSTACONFIRMATION
            EventType_t eventType;              // CSTA_SET_MWI_CONF
} ACSEventHeader_t;

typedef struct CSTASetMwiConfEvent_t {
    Nulltype      null;
} CSTASetMwiConfEvent_t;

typedef struct
{
            ACSEventHeader_t eventHeader;
            union
            {
            struct
            {
                        InvokeID_t  invokeID;
                        union
                        {
                        CSTASetMwiConfEvent_t setMwi;
                        } u;
                        } cstaConfirmation;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Chapter 8: Query Service Group

Query Service Group describes services that allow a client application to query the switch to provide the state of device features and static attributes of a device. The following sections describes the Query services that Application Enablement Services (AE Services) supports.

Query ACD Split Service

Summary

- Direction: Client to Switch
- Function: cstaEscapeService() Confirmation Event: CSTAEscapeServiceConfEvent
- Private Data Function: attQueryACDSplit()
- Private Data Confirmation Event: ATTQueryACDSplitConfEvent
- Service Parameters: noData
- Private Parameters: device
- Ack Parameters: noData
- Ack Private Parameters: availableAgents, callsInQueue, agentsLoggedIn
- Nak Parameter: universalFailure

Functional Description

The Query ACD Split service provides the number of ACD agents available to receive calls through the split, the number of calls in queue, and the number of agents logged in. The number of calls in queue does not include direct-agent calls.

Service Parameters

noData	None for this service.

Private Parameters:

device	[mandatory] Must be a valid ACD split extension.

Ack Parameters:

noData	None for this service.

Ack Private Parameters:

availableAgents	[mandatory] Specifies the number of ACD agents available to receive calls through the specified split.
callsInQueue	[mandatory] Specifies the number of calls in queue (not including direct-agent calls).
agentsLoggedIn	[mandatory] Specifies the number of ACD agents logged in.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

None

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
       ACSHandle_t acsHandle,
       InvokeID_t invokeID,
       PrivateData_t *privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
       ACSHandle_t  acsHandle;
       EventClass_t eventClass;// CSTACONFIRMATION
       EventType_t eventType;// CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
       ACSEventHeader_t eventHeader;
       union
       {
            struct
            {
                 InvokeID_t invokeID;
                 union
                 {
                      CSTAEscapeSvcConfEvent escapeService;
                 }u;
            } cstaConfirmation;
       } event;
       char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t {
       Nulltype null
} CSTAEscapeSvcConfEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryACDSplit() - Service Request Private Data Setup Function

RetCode_t*attQueryACDSplit (// returns NULL if no
// parameter specified
       ATTPrivateData_t *privateData,
       DeviceID_t *device);

typedef struct ATTPrivateData_t {
       char         vendor[32];
       unsigned short length;
       char         data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;


// ATTQueryACDSplitConfEvent - Service Response Private Data

typedef struct
{
       ATTEventType eventType;// ATT_QUERY_ACD_SPLIT_CONF
       union
       {
            ATTQueryACDSplitConfEvent_t queryACDSplit;
       }u;
char   heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTQueryACDSplitConfEvent_t
{
       short availableAgents;// number of available agents
// to receive call
       short callsInQueue;// number of calls in queue
       short agentsLoggedIn;// number of agents logged in
} ATTQueryACDSplitConfEvent_t;
```

Query Agent Login Service

Summary

- Direction: Client to Switch
- Function: cstaEscapeService()
- Confirmation Event: CSTAPrivateEvent, CSTAEscapeServiceConfEvent
- Private Data Function: attQueryAgentLogin(), ATTQueryAgentLoginResp()
- Private Data Confirmation Event: ATTQueryAgentLoginConfEvent
- Service Parameters: noData
- Private Parameters: device
- Ack Parameters: noData
- Ack Private Parameters: privEventCrossRefID
- Private Event Parameters: privEventCrossRefID, list
- Nak Parameter: universalFailure

Functional Description

The Query Agent Login Service provides the extension of each ACD agent logged into the specified ACD split. This service is unlike most other services because the confirmation event provides a unique private event cross reference ID that associates a subsequent CSTAPrivateEvent (containing the actual ACD agent login data) with the original request. The private event cross reference ID is the only data returned in the confirmation event. After returning the confirmation event, the service returns a sequence of CSTAPrivateEvents. Each CSTAPrivateEvent contains the private event cross reference ID, and a list. The list contains the number of extensions in the message, and up to 10 extensions of ACD agents logged into the ACD split.

The entire sequence of CSTAPrivateEvents may contain a large volume of information (up to the maximum number of logged-in agents allowed in an ACD Split). The service provides the private event cross reference ID in case an application has issued multiple Query Agent Login requests. The final CSTAPrivateEvent specifies that it contains zero extensions and serves to inform the application that no more messages will be sent in response to this query.

Service Parameters:

noData	None for this service.

Private Parameters:

device	[mandatory] Must be a valid ACD split extension.

Ack Parameters:

noData	None for this service.

Ack Private Parameters:

privEventCrossRefID	Contains a unique handle to identify subsequent CSTAPrivateEvents with this request.

Private Event Parameters:

privEventCrossRefID	[mandatory] The handle to the query agent login request for which this CSTAPrivateEvent is reported.
list	[mandatory] A list structure with the following information: the count (0 - 10) of how many extensions are in the message and an array of up to 10 extensions. A count value of 0 is like an "end of file" - i.e., there are no additional CSTAPrivateEvents for the query.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

- A single Query Agent Login Request may result in multiple CSTAPrivateEvents returned to the client after the return of the confirmation event. All messages are contained in private data of the CSTAPrivateEvents.
- This service uses a private event cross reference ID to provide a way for clients to correlate incoming CSTAPrivateEvents with an original Query Agent Login request.
- Each separate CSTAPrivateEvent may contain up to 10 extensions.

- Each separate CSTAPrivateEvent contains a number indicating how many extensions are in the message. The last CSTAPrivateEvent has the number set to zero.
- The service receives each response message from the switch and passes it to the application in a CSTAPrivateEvent. The application must be prepared to receive and deal with a potentially large number of extensions received in multiple CSTAPrivateEvents after it receives the confirmation event.

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      PrivateData_t*privateData);


// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAEscapeSvcConfEvent_tescapeService;
              }u;
          } cstaConfirmationEvent;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t {
      Nulltypenull
} CSTAEscapeSvcConfEvent_t;
```

Syntax (Continued)

```
// CSTAPrivateEvent - Private event for reporting data

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {

              union
              {
                  CSTAPrivateEvent_tprivateData;
              }u;
          } cstaEventReport;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAPrivateEvent_t {
      Nulltypenull
} CSTAPrivateEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryAgentLogin() - Service Request Private Data Setup Function

RetCode_tattQueryAgentLogin ( // returns NULL if no
// parameter specified
      ATTPrivateData_t*privateData,
      DeviceID_t*device);

typedef struct ATTPrivateData_t {
      char        vendor[32];
      unsigned shortlength;
      char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTQueryAgentLoginConfEvent - Confirmation Event Private Data

typedef struct
{
      ATTEventTypeeventType;// ATT_QUERY_AGENT_LOGIN_CONF
      union
      {
          ATTQueryAgentLoginConfEvent_tqueryAgentLogin;

      }u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTQueryAgentLoginConfEvent_t {
      ATTPrivEventCrossRefID_tprivEventCrossRefID;
} ATTQueryAgentLoginConfEvent_t;

// ATTQueryAgentLoginEvent - Private Event Private Data
typedef struct
{
      ATTEventTypeeventType;// ATT_QUERY_AGENT_LOGIN_RESP
      union
      {
          ATTQueryAgentLoginResp_tqueryAgentLoginResp;
      }u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;
```

Private Parameter Syntax (Continued)

```
typedef struct ATTQueryAgentLoginResp_t
{
      ATTPrivEventCrossRefID_tprivEventCrossRefID;
                  // cross reference ID
      struct {
          shortcount;// number of extensions in
                  // device[]
          DeviceID_tdevice[10];// up to 10 extensions
      } list;
} ATTQueryAgentLoginResp_t;
```

Query Agent State Service

Summary

- Direction: Client to Switch
- Function: cstaQueryAgentState()
- Confirmation Event: CSTAQueryAgentStateConfEvent
- Private Data Function: attQueryAgentState()
- Private Data Confirmation Event: ATTQueryAgentStateConfEvent (private data version 6 and 7), ATTV5QueryAgentStateConfEvent (private data version 5), ATTV4QueryAgentStateConfEvent (private data versions 2-4)
- Service Parameters: device
- Private Parameters: split
- Ack Parameters: agentState
- Ack Private Parameters: workMode, talkState, reasonCode, pendingWorkMode, pendingReasonCode
- Nak Parameter: universalFailure

Functional Description:

This service provides the agent state of an ACD agent. The agent's state is returned in the CSTA AgentState parameter. The private talkState parameter indicates if the agent is idle or busy. The private workMode parameter has the agent's work mode as defined by the DEFINITY PBX. The private reasonCode has the agent's reasonCode if one is set. The private pendingWorkMode and pendingReasonCode have the work mode and reason code that will take effect as soon as the agent's current call is terminated.

Service Parameters:

device	[mandatory] Must be a valid agent extension or a logical agent ID.

Private Parameters:

split	[optional] If specified, it must be a valid ACD split extension. This parameter is optional in an EAS environment, but it is mandatory for a non-EAS environment.

Ack Parameters:

agentState [mandatory - partially supported] The ACD agent state. Agent state will be one of the following values:

- AG_NULL - The agent is logged out of the device/ACD split.
- AG_NOT_READY - The agent is occupied with some task other than that of serving a call.
- AG_WORK_NOT_READY - The agent is occupied with after call work. The agent should not receive additional ACD calls in this state.
- AG_READY - The agent is available to accept calls or is currently busy with an ACD call.
- The DEFINITY PBX does not support the AG_WORK_READY state.

Ack Private Parameters:

workMode [optional] This parameter provides the agent work mode as defined by the DEFINITY PBX. Valid values include:

- WM_AUTO_IN Indicates that the agent is allowed to receive a new call immediately after disconnecting from the previous call. The talkState parameter indicates whether the agent is busy or idle.
- WM_MANUAL_IN Indicates that the agent is automatically changed to the WM_AFTCAL_WK state immediately after disconnecting from the previous call.
- WM_ AFTCAL_WK Indicates that the agent is in the WM_AFTCAL_WK mode. (A query agent state on an agent in the WM_AFTCAL_WK state returns an agentState parameter value of AG_WORK_NOT_READY.)
- WM_AUX_WORK Indicates that the agent is in the WM_ AUX_WORK mode. (A query agent state on an agent in the WM_ AUX_WORK state returns an agentState parameter value of AG_ NOT_READY.)

talkState [optional] The talkState parameter provides the actual readiness of the agent. Valid values are:

- TS_ON_CALL Indicates that the agent is occupied with serving a call
- TS_IDLE Indicates that the agent is ready to accept calls.

reasonCode [optional] Specifies the reason for change of work mode to WM_AUX_WORK or the logged-out (AM_LOG_OUT) state.

For private data version 7 valid reason codes range from 0 to 99. A value of 0 indicates that the reason code is not available. The meaning of the codes 1 through 99 is defined by the application. This range of reason codes is supported by private data version 7 only.

Private data versions 6 and 5 support single digit reason codes 1 through 9. A value of 0 indicates that the reason code is not available. The meaning of the code (1-9) is defined by the application.

Private data version 4 and earlier do not support reason codes.

pendingWorkMode [optional] Specifies the work mode which will take effect when the agent gets off the call. If no work mode is pending then pendingWorkMode will be set to WM_NONE (-1).

pendingReasonCode [optional] Specifies the pending reason code which will take effect when the agent gets off the call. A value of 0 indicates that the pending reason code is not available.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

- Communication Manager does not support the AG_WORK_READY state for agentState.
- Except agentState of AG_NULL, all confirmation includes private parameters of agent workMode and talkState. The actual readiness of the agent depends on values for these private parameters. In particular, the value for talkState determines if the agent is busy on a call or ready to accept calls.
- The Communication Manager Agent Work Mode to CSTA Agent State Mapping is as follows:

Communication Manager Agent Work Mode	CSTA Agent State (workMode)
Agent not logged in	NULL
WM_AUX_WORK	AG_NOT_READY
WM_AFTCAL_WORK	AG_WORK_NOT_READY
WM_AUTO_IN	AG_READY (workMode=WM_AUTO_IN)
WM_MANUAL_IN	AG_READY (workMode=WM_MANUAL_IN)

- If the agent workMode is WM_AUTO_IN, the Query Agent State service always returns AG_READY. The agent is immediately made available to receive a new call after disconnecting from the previous call.

Agent Activity	agentState	talkState
Ready to accept calls	AG_READY	TS_IDLE
Occupied with a call	AG_READY	TS_ON_CALL
Disconnected from call	AG_READY	TS_IDLE

Syntax

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

// cstaQueryAgentState() - Service Request

RetCode_t   cstaQueryAgentState (
       ACSHandle_tacsHandle,
       InvokeID_tinvokeID,
       DeviceID_t*device,
       PrivateData_t*privateData);

// CSTAQueryAgentStateConfEvent - Service Response

typedef struct
{
       ACSHandle_t acsHandle;
       EventClass_teventClass;// CSTACONFIRMATION
       EventType_teventType;// CSTA_QUERY_AGENT_STATE_CONF
} ACSEventHeader_t;

typedef struct
{
       ACSEventHeader_teventHeader;
       union
       {
            struct
            {
                 InvokeID_tinvokeID;
                 union
                 {
                      CSTAQueryAgentStateConfEvent_t
       queryAgentState;
                 }u;
            } cstaConfirmation;
       } event;
char   heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAQueryAgentStateConfEvent_t {
       AgentState_tagentState;
} CSTAQueryAgentStateConfEvent_t;

typedef enum AgentState_t {
       AG_NOT_READY = 0,
       AG_NULL  = 1,
       AG_READY = 2,
       AG_WORK_NOT_READY = 3,
       AG_WORK_READY = 4// not used in G3 CSTA
} AgentState_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryAgentState() - Service Request Private Data Setup Function

RetCode_tattQueryAgentState (
      ATTPrivateData_t*privateData,
      DeviceID_t*split);

typedef struct ATTPrivateData_t
{
      char        vendor[32];
      unsigned shortlength;
      char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTQueryAgentStateConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
      ATTEventType_teventType;// ATT_QUERY_AGENT_STATE_CONF
      union
      {
          ATTQueryAgentStateConfEvent_tqueryAgentState;
      } u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTQueryAgentStateConfEvent_t
{
      ATTWorkMode_tworkMode;// agent work mode
      ATTTalkState_ttalkState;// agent talk state
      long        reasonCode;// single digit 1-9
      ATTWorkMode_tpendingWorkMode;// pending agent work mode
      long        pendingReasonCode;// single digit 1-9
} ATTQueryAgentStateConfEvent_t;

typedef enum ATTWorkMode_t
{
      WM_NONE = -1,   // No workmode is pending
      WM_AUX_WORK = 1,
      WM_AFTCAL_WK = 2,
```

Private Data Version 6 (Continued)

```
      WM_AUTO_IN = 3,
      WM_MANUAL_IN = 4
} ATTWorkMode_t;

typedef enum ATTTalkState_t
{
      TS_ON_CALL = 0,
      TS_IDLE = 1
} ATTTalkState_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryAgentState() - Service Request Private Data Setup Function

RetCode_tattQueryAgentState (
       ATTPrivateData_t*privateData,
       DeviceID_t*split);

typedef struct ATTPrivateData_t
{
       char         vendor[32];
       unsigned shortlength;
       char         data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTV5QueryAgentStateConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
       ATTEventType_teventType;// ATTV5_QUERY_AGENT_STATE_CONF
       union
       {
           ATTV5QueryAgentStateConfEvent_tv5queryAgentState;
       } u;
char   heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTV5QueryAgentStateConfEvent_t
{
       ATTWorkMode_tworkMode;// agent work mode
       ATTTalkState_ttalkState;// agent talk state
       long         reasonCode;// single digit 1-9
} ATTV5QueryAgentStateConfEvent_t;

typedef enum ATTWorkMode_t
{
       WM_AUX_WORK = 1,
       WM_AFTCAL_WK = 2,
       WM_AUTO_IN = 3,
       WM_MANUAL_IN = 4
} ATTWorkMode_t;

typedef enum ATTTalkState_t
{
       TS_ON_CALL = 0,
       TS_IDLE  = 1
} ATTTalkState_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryAgentState() - Service Request Private Data Setup Function

RetCode_tattQueryAgentState (
       ATTPrivateData_t*privateData,
       DeviceID_t*split);

typedef struct ATTPrivateData_t
{
       char         vendor[32];
       unsigned shortlength;
       char         data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTV4QueryAgentStateConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
       ATTEventType_teventType;// ATTV4_QUERY_AGENT_STATE_CONF
       union
       {
            ATTV4QueryAgentStateConfEvent_tv4queryAgentState;
       } u;
char   heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTV4QueryAgentStateConfEvent_t
{
       ATTWorkMode_tworkMode;// agent work mode
       ATTTalkState_ttalkState;// agent talk state
} ATTV4QueryAgentStateConfEvent_t;

typedef enum ATTWorkMode_t
{
       WM_AUX_WORK = 1,
       WM_AFTCAL_WK = 2,
       WM_AUTO_IN = 3,
       WM_MANUAL_IN = 4
} ATTWorkMode_t;

typedef enum ATTTalkState_t
{
       TS_ON_CALL = 0,
       TS_IDLE  = 1
} ATTTalkState_t;
```

Query Call Classifier Service

Summary

- Direction: Client to Switch
- Function: cstaEscapeService()
- Confirmation Event: CSTAEscapeServiceConfEvent
- Private Data Function: attQueryCallClassifier()
- Private Data Confirmation Event: ATTQueryCallClassifierConfEvent
- Service Parameters: noData
- Private Parameters: noData
- Ack Parameters: noData
- Ack Private Parameters: numAvailPorts, numInUsePorts
- Nak Parameter: universalFailure

Functional Description:

This service provides the number of "idle" and "in-use" ports (e.g., TN744). The "in use" number is a snapshot of the call classifier port usage.

Service Parameters:

noData	None for this service.

Private Parameters:

noData	None for this service.

Ack Parameters:

noData	None for this service.

Ack Private Parameters:

numAvailPorts	[mandatory] The number of available ports.
numInUsePorts	[mandatory] The number of "in use" ports.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786

Detailed Information:

None

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      PrivateData_t*privateData);


// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAEscapeSvcConfEvent_tescapeService;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t {
      Nulltypenull
} CSTAEscapeSvcConfEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryCallClassifier() - Service Request Private Data Setup
Function

RetCode_tattQueryCallClassifier (// no private parameter,
// but must be called
      ATTPrivateData_t*privateData);

typedef struct ATTPrivateData_t {
      char    vendor[32];
      ushortlength;
      char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTQueryCallClassifierConfEvent - Service Response Private Data

typedef struct
{
      ATTEventType_teventType;// ATT_QUERY_CALL_CLASSIFIER_CONF
      union
      {
          ATTQueryCallClassifierConfEvent_tqueryCallClassifier;
      }u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;


typedef struct ATTQueryCallClassifierConfEvent_t
{
      shortnumAvailPorts;// number of available ports
      shortnumInUsePorts;// number of in use ports
} ATTQueryCallClassifierConfEvent_t;
```

Query Device Info

Summary

- Direction: Client to Switch
- Function: cstaQueryDeviceInfo()
- Confirmation Event: CSTAQueryDeviceInfoConfEvent
- Private Data Confirmation Event: ATTQueryDeviceInfoConfEvent (private data version 5), ATTV4QueryDeviceInfoConfEvent (private data versions 2-4)
- Service Parameters: device
- Ack Parameters: device, deviceType, deviceClass
- Ack Private Parameters: extensionClass, associatedDevice, associatedClass
- Nak Parameter: universalFailure

Functional Description:

This service provides the class and type of a device. The class is one of the following attributes: voice, data, image, or other. The type is one of the following attributes: station, ACD, ACD Group, or other. The G3 Extension class is provided in the CSTA private data.

Service Parameters:

device	[mandatory] Must be a valid on-PBX station extension.

Ack Parameters:

device	[optional - supported] Normally this is the same ID specified in the device parameter in the request. See associatedDevice and associatedClass below.
deviceType	[mandatory] The device type (mapped from G3 extension class).
deviceClass	[mandatory] The device class (mapped from G3 extension class).

Ack Private Parameters:

extensionClass	[mandatory] The G3 Extension Class for the device.
associatedDevice	[optional] If the device specified in the request is a physical device of a logical agent who is logged in, the logical ID of that agent is returned in this parameter. Vice-versa, if the device specified in the request is the logical ID of a logged-in agent, the physical device ID of that agent is returned in this parameter. Otherwise, a null string is returned. This parameter is supported by private data version 5 and later only.
associatedClass	[optional] The G3 Extension Class for the associatedDevice. It is EC_LOGICAL_AGENT, If the associatedDevice is a device ID of a logical agent; otherwise it has the value of EC_OTHER. This parameter is supported by private data version 5 and later only.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

The deviceType and deviceClass parameters are mapped from the G3 extension class as follows:

G3 Extension Class	CSTA Device Class	CSTA Device Type
VDN	Voice[1]	ACD Group
Hunt Group (ACD Split)	Voice	ACD Group
Announcement	Voice	Other
Data extension	Data	Station
Voice extension - Analog	Voice	Station
Voice extension - Proprietary	Voice	Station
Voice extension - BRI	Voice	Station
Logical Agent	Voice	Other
CTI	Data	Other
Other (modem pool, etc.)	Other	Other

1. There is an additional private data qualifier that indicates if it is a VDN.

Syntax

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

// cstaQueryDeviceInfo() - Service Request

RetCode_t   cstaQueryDeviceInfo (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      DeviceID_t*device,
      PrivateData_t*privateData);

// CSTAQueryDeviceInfoConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_QUERY_DEVICE_INFO_CONF
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAQueryDeviceInfoConfEvent_t
      queryDeviceInfo;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAQueryDeviceInfoConfEvent_t {
      DeviceID_tdevice;
      DeviceType_tdeviceType;
      DeviceClass_tdeviceClass;
} CSTAQueryDeviceInfoConfEvent_t;
```

Syntax (Continued)

```
// Device Types
typedef enum DeviceType_t {
      DT_STATION = 0,
      DT_LINE = 1,// not an expected G3 response
      DT_BUTTON = 2, // not an expected G3 response
      DT_ACD = 3,
      DT_TRUNK = 4, // not an expected G3 response
      DT_OPERATOR = 5, // not an expected G3 response
      DT_STATION_GROUP = 16, // not an expected G3 response
      DT_LINE_GROUP = 17, // not an expected G3 response
      DT_BUTTON_GROUP = 18, // not an expected G3 response
      DT_ACD_GROUP = 19,
      DT_TRUNK_GROUP = 20, // not an expected G3 response
      DT_OPERATOR_GROUP = 21,// not an expected G3 response
      DT_OTHER = 255

} DeviceType_t;

typedef unsigned charDeviceClass_t;

// Device Classes
#define           DC_VOICE   0x80
#define           DC_DATA    0x40
#define           DC_IMAGE   0x20// not an expected G3 response
#define           DC_OTHER   0x10
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTQueryDeviceInfoConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
      ATTEventType_teventType;// ATT_QUERY_DEVICE_INFO_CONF
      union
      {
          ATTQueryDeviceInfoConfEvent_tqueryDeviceInfo;
      } u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTQueryDeviceInfoConfEvent_t
{
      ATTExtensionClass_textensionClass;
      ATTExtensionClass_tassociatedClass;
      DeviceID_t  associatedDevice;
} ATTQueryDeviceInfoConfEvent_t;

typedef enum ATTExtensionClass_t
{
      EC_VDN  = 0,
      EC_ACD_SPLIT = 1,
      EC_ANNOUNCEMENT = 2,
      EC_DATA = 4,
      EC_ANALOG = 5,
      EC_PROPRIETARY = 6,
      EC_BRI  = 7,
      EC_CTI  = 8,
      EC_LOGICAL_AGENT= 9,
      EC_OTHER = 10
} ATTExtensionClass_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV4QueryDeviceInfoConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
      ATTEventType_teventType;// ATTV4_QUERY_DEVICE_INFO_CONF
      union
      {
          ATTV4QueryDeviceInfoConfEvent_tv4queryDeviceInfo;
      } u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTV4QueryDeviceInfoConfEvent_t
{
      ATTExtensionClass_textensionClass;
} ATTV4QueryDeviceInfoConfEvent_t;

typedef enum ATTExtensionClass_t
{
      EC_VDN  = 0,
      EC_ACD_SPLIT = 1,
      EC_ANNOUNCEMENT = 2,
      EC_DATA = 4,
      EC_ANALOG = 5,
      EC_PROPRIETARY = 6,
      EC_BRI  = 7,
      EC_CTI  = 8,
      EC_LOGICAL_AGENT= 9,
      EC_OTHER = 10
} ATTExtensionClass_t;
```

Query Device Name Service

Summary

- Direction: Client to Switch
- Function: cstaEscapeService()
- Confirmation Event: CSTAEscapeSvcConfEvent
- Private Data Function: attQueryDeviceName()
- Private Data Confirmation Event: ATTQueryDeviceNameConfEvent (private data version 7), ATTV6QueryDeviceNameConfEvent (private data versions 5 and 6), ATTV4QueryDeviceNameConfEvent (private data versions 2-4)
- Service Parameters: noData
- Private Parameters: device
- Ack Parameters: noData
- Ack Private Parameters: deviceType, device, name, uname
- Nak Parameter: universalFailure

Functional Description:

The Query Device Name service allows an application to query the switch with an extension of a device and receive the associated name of the device. The name is retrieved from the Communication Manager Integrated Directory Database.

This service will allow an application to identify the names administered in Communication Manager with device extension numbers without maintaining its own database.

Service Parameters:

noData	None for this service.

Private Parameters:

device	[mandatory] Must be a valid device extension.

Ack Parameters:

noData	None for this service.

Ack Private Parameters:

deviceType	[mandatory] Specifies the device type of the device: ● DT_ACD_SPLIT - ACD Split (Hunt Group) ● DT_ANNOUNCEMENT - announcement ● DT_DATA - data extension ● DT_LOGICAL_AGENT - logical agent ● DT_STATION - station extension ● DT_TRUNK_ACCESS_CODE - Trunk Access Code ● DT_VDN - VDN
other	If no name is administered in Communication Manager for the device, the attQueryDeviceName() value query response returns the information that the switch is providing. In this case, when the switch returns "other" (0x18) as the domainType in the value query response, the confirmation event sets the following parameters. **deviceType**, **device**, and **uname**
device	[mandatory] Specifies the extension number of the device. **NOTE:** If no name is administered in Communication Manager for the device, the attQueryDeviceName() value query response returns the information that the switch is providing. In this case, when the switch returns "other" (0x18) as the domainType in the value query response, the confirmation event sets ***device*** to either the extension or trunk access code (tac) of the device, (depending on the domain type)

name [mandatory] Specifies the associated name of the device. This is a string of 1-15 ASCII characters for private data version 3 and 4. This is a string of 1-27 ASCII characters for private data version 5 and later only.

NOTE: If no name is administered in Communication Manager for the device, the attQueryDeviceName() value query response returns the information that the switch is providing. In this case, when the switch returns "other" (0x18) as the domainType in the value query response, the confirmation event sets ***name*** to the name returned (if it is not null) If the name is null, then null is returned.

The name of a device is administered in Communication Manager. Non-standard 8-bit OPTREX characters supported on the displays of the 84xx series terminals may be reported in name parameter. The 84xx terminal displays supports a limited number of non-standard characters (in addition to the standard 7-bit ASCII display characters), including Katakana, graphical characters, and Eurofont (European-type) characters. The tilde,~, character is not defined in the OPTREX set and is used as the toggle character (turn on/off 8-bit character set) to indicate subsequent characters are to have the high-bit set (turned off by a following ~ character, if any). If non-standard 8-bit OPTREX characters are administered in the switch for the device, then the tilde,~, character will be reported in its name. An application needs to map the non-standard 8-bit OPTREX characters to its proper printable characters.

uname [mandatory] Specifies the associated name of the device in Unicode . This parameter is supported by private data version 5 and later only.

NOTE: If no name is administered in Communication Manager for the device, the attQueryDeviceName() value query response returns the information that the switch is providing. In this case, when the switch returns "other" (0x18) as the domainType in the value query response, the confirmation event sets ***uname*** to uname hex values.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786

- VALUE_OUT_OF_RANGE (3) (CS0/100) Invalid parameter value specified.
- OBJECT_NOT_KNOWN (4) (CS0/96) Mandatory parameter is missing.
- INVALID_CSTA_DEVICE_IDENTIFIER (12) (CS0/28) An invalid device identifier has been specified in device.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) (CS0/50) This service is requested on a switch administered as a release earlier then G3V4.

Detailed Information:

- Incomplete Names - The names returned by this service may not be the full names since they are limited to 15 characters in the Integrated Directory database.
- Security - G3 Switch does not provide security mechanisms for this service.
- Traffic Control - The application is responsible for controlling the message traffic on the CTI link. An application should minimize traffic by requesting device names only when needed. This service is not intended for use by an application to create its own copy of the Integrated Directory database. If the number of outstanding requests reaches the switch limit, the response time may be as long as 30 seconds.

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      PrivateData_t*privateData);

// CSTAEscapeSvcConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_ESCAPE_SVC_CONF
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAEscapeSvcConfEventescapeService;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t {
      Nulltypenull
} CSTAEscapeSvcConfEvent_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryDeviceName() - Service Request Private Data Setup Function

RetCode_tattQueryDeviceName (
      ATTPrivateData_t*privateData,
      DeviceID_t*device);

typedef struct ATTPrivateData_t
{
      char    vendor[32];
      ushortlength;
      char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTQueryDeviceNameConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
      ATTEventType_teventType;// ATT_QUERY_DEVICE_NAME_CONF
      union
      {
          ATTQueryDeviceNameConfEvent_tqueryDeviceName;
      } u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTQueryDeviceNameConfEvent_t
{
      ATTDeviceType_tdeviceType;
      DeviceID_t  device;
      DeviceID_t  name;  // 1–27 ASCII character string
      ATTUnicodeDeviceID_tuname; // name in Unicode
} ATTQueryDeviceNameConfEvent_t;

typedef enum ATTDeviceType_t
{
      ATT_DT_ACD_SPLIT= 1,
      ATT_DT_ANNOUNCEMENT= 2,
      ATT_DT_DATA     = 3,
      ATT_DT_LOGICAL_AGENT= 4,
      ATT_DT_STATION= 5,
      ATT_DT_TRUNK_ACCESS_CODE= 6,
      ATT_DT_VDN      = 7
}ATTDeviceType_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryDeviceName() - Service Request Private Data Setup Function

RetCode_tattQueryDeviceName (
      ATTPrivateData_t*privateData,
      DeviceID_t*device);

typedef struct ATTPrivateData_t
{
      char    vendor[32];
      ushortlength;
      char    data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTV4QueryDeviceNameConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
      ATTEventType_teventType;// ATTV4_QUERY_DEVICE_NAME_CONF
      union
      {
          ATTV4QueryDeviceNameConfEvent_tv4queryDeviceName;
      } u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTV4QueryDeviceNameConfEvent_t
{
      ATTDeviceType_tdeviceType;
      DeviceID_tdevice;
      char        name[16]; // 1–15 ASCII character string
} ATTV4QueryDeviceNameConfEvent_t;

typedef enum ATTDeviceType_t
{
      ATT_DT_ACD_SPLIT= 1,
      ATT_DT_ANNOUNCEMENT= 2,
      ATT_DT_DATA     = 3,
      ATT_DT_LOGICAL_AGENT= 4,
      ATT_DT_STATION= 5,
      ATT_DT_TRUNK_ACCESS_CODE= 6,
      ATT_DT_VDN      = 7
} ATTDeviceType_t;
```

Query Do Not Disturb Service

Summary

- Direction: Client to Switch
- Function: cstaQueryDoNotDisturb()
- Confirmation Event: CSTAQueryDoNotDisturbConfEvent
- Service Parameters: device
- Ack Parameters: doNotDisturb
- Nak Parameter: universalFailure

Functional Description:

This service provides the status of the send all calls feature expressed as on or off at a device. The status will always be reported as off when the extension does not have a coverage path.

Service Parameters:

device	[mandatory] Must be a valid on-PBX station extension that supports the send all calls (SAC) feature.

Ack Parameters:

doNotDisturb	[mandatory] Status of the send all calls feature expressed as on (TRUE) or off (FALSE).

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

None

Syntax

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

// cstaQueryDoNotDisturb() - Service Request

RetCode_t   cstaQueryDoNotDisturb (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      DeviceID_t*device,
      PrivateData_t*privateData);

// CSTAQueryDoNotDisturbConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;
      EventType_teventType;
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAQueryDndConfEvent_tqueryDnd;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;


typedef struct CSTAQueryDndConfEvent_t {
      Boolean_tdoNotDisturb;// TRUE = on, FALSE = off
} CSTAQueryDndConfEvent_t;
```

Query Forwarding Service

Summary

- Direction: Client to Switch
- Function: cstaQueryForwarding()
- Confirmation Event: CSTAQueryForwardingConfEvent
- Service Parameters: device
- Ack Parameters: forward
- Nak Parameter: universalFailure

Functional Description:

This service provides the status and forward-to-number of the Call Forwarding feature for a device. The status is expressed as on or off. Communication Manager supports only one Forwarding Type (Immediate). Thus, the on/off indicator is only specified for the Immediate type. The Call Forwarding feature may by turned on for many types (G3 redirection Criteria), and the actual forward type is dependent on how the feature is administered in Communication Manager.

Service Parameters:

device	[mandatory] Must be a valid on-PBX station extension that supports the Call Forwarding feature.

Ack Parameters:

forward	[mandatory] This is a list of forwarding parameters. The list contains a count of how many items are in the list. Since Communication Manager stores only one forwarding address, the count is one. Each element in the list contains the following: forwardingType, forwardingOn, and forwardDN. For Communication Manager, forwardingType will always be FWD_IMMEDIATE; forwardingOn will indicate (on/off(status (TRUE indicates on, FALSE indicates off); and forwardDN will contain the forward-to-number.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

Communication Manager supports only one CSTA Forwarding Type: Immediate. Thus, each response contains information for the Immediate type.

Syntax

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

// cstaQueryForwarding() - Service Request

RetCode_t   cstaQueryForwarding (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      DeviceID_t*deviceID,
      PrivateData_t*privateData);

// CSTAQueryForwardingConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_QUERY_FWD_CONF
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAQueryFwdConfEvent_tqueryFwd;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAQueryFwdConfEvent_t {
      ListForwardParameters_tforward;
} CSTAQueryFwdConfEvent_t;
```

Syntax (Continued)

```
typedef struct ListForwardParameters_t {
      shortcount;         // only 1 is provided in list
      ForwardingInfo_tparam[7];
} ListForwardParameters_t;

typedef struct ForwardingInfo_t {
      ForwardingType_tforwardingType; // FWD_IMMEDIATE
      Boolean forwardingOn;// TRUE = on, FALSE = off
      DeviceID_tforwardDN;
} ForwardingInfo_t;

typedef enum ForwardingType_t {
      FWD_IMMEDIATE = 0,// only type supported
      FWD_BUSY = 1,// not supported
      FWD_NO_ANS = 2,// not supported
      FWD_BUSY_INT = 3,// not supported
      FWD_BUSY_EXT = 4,// not supported
      FWD_NO_ANS_INT = 5,// not supported
      FWD_NO_ANS_EXT = 6// not supported
} ForwardingType_t;
```

Query Message Waiting Service

Summary

- Direction: Client to Switch
- Function: cstaQueryMsgWaitingInd()
- Confirmation Event: CSTAQueryMwiConfEvent
- Private Data Confirmation Event: ATTQueryMwiConfEvent
- Service Parameters: device
- Ack Parameters: messages
- Ack Private Parameters: applicationType
- Nak Parameter: universalFailure

Functional Description:

The Query Message Waiting Service provides status of the message waiting indicator expressed as on or off for a device. The applications that turn the indicator on (that is, ASAI, Property Management, Message Center, Voice Processing, Leave Word Calling) are reported in the private data.

Service Parameters:

device	[mandatory] Must be a valid on-PBX station extension that supports the Message Waiting Indicator (MWI) feature.

Ack Parameters:

messages	[mandatory] Indicates the on/off status (TRUE indicates on, FALSE indicates off) of the MWI for this device.

Ack Private Parameters:

applicationType	[mandatory] Indicates the applications that turned on the MWI for the device

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786

INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

- Application Type - The private data member applicationType is a bit map where one bit is set for each application that turned on the indicator. Multiple applications may turn on the indicator. The applications represented are: CTI/ASAI, Property Management (PMS), Message Center (MCS), Voice Messaging, and Leave Word Calling (LWC).

  To find out which applications turned on the indicator, the application must use a bit mask as shown in the following table:

bit:	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
Application	N/A	N/A	N/A	CTI/ASAI	LWC	PMS	Voice	MCS

- Setting MWI Status - An application can set the MWI status through the CSTASetMsgWaitingInd Service.
- System Starts - System cold starts cause the switch to lose the MWI status. Other types of restart do not affect the MWI status.

Syntax

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

// cstaQueryMsgWaitingInd() - Service Request

RetCode_t   cstaQueryMsgWaitingInd (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      DeviceID_t*device,
      PrivateData_t*privateData);

// CSTAQueryMsgWaitingIndConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_QUERY_MWI_CONF
} ACSEventHeader_t;

 typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAQueryMwiConfEvent_tqueryMwi;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAQueryMwiConfEvent_t {
      Booleanmessages;// TRUE = on, FALSE = off
} CSTAQueryMwiConfEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTQueryMwiConfEvent - Service Response Private Data

typedef struct
{
      ATTEventTypeeventType;// ATT_QUERY_MWI_CONF
      union
      {
          ATTQueryMwiConfEvent_tqueryMwi;
      }u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTQueryMwiConfEvent_t
{
      ATTMwiApplication_tapplicationType;// application type
} ATTQueryMwiConfEvent_t;

typedefunsigned charATTMwiApplication_t;
#define           AT_MCS  0x01// bit 1
#define           AT_VOICE0x02// bit 2
#define           AT_PROPMGT0x04 // bit 3
#define           AT_LWC  0x08// bit 4
#define           AT_CTI  0x10// bit 5
```

Query Station Status Service

Summary

- Direction: Client to Switch
- Function: cstaEscapeService()
- Confirmation Event: CSTAEscapeServiceConfEvent
- Private Data Function: attQueryStationStatus()
- Private Data Confirmation Event: ATTQueryStationStatusConfEvent
- Service Parameters: noData
- Private Parameters: device
- Ack Parameters: noData
- Ack Private Parameters: stationStatus
- Nak Parameter: universalFailure

Functional Description:

The Query Station Status service provides the idle and/or busy state of a station. The "busy" state is returned if the station is active with a call. The "idle" state is returned if the station is not active with any call.

Service Parameters:

noData	None for this service.

Private Parameters:

device	[mandatory] Must be a valid station device.

Ack Parameters:

noData	None for this service.

Ack Private Parameters:

stationStatus	[mandatory] Specifies the busy/idle state (TRUE indicates busy, FALSE indicates idle) of the station.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

None

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      PrivateData_t*privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAEscapeSvcConfEvent_tescapeService;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t {
      Nulltypenull
} CSTAEscapeSvcConfEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryStationStatus() - Service Request Private Data Setup
Function

RetCode_tattQueryStationStatus (// returns NULL if no
// parameter specified
      ATTPrivateData_t*privateData,
      DeviceID_t*device);

typedef struct ATTPrivateData_t {
      char        vendor[32];
      unsigned shortlength;
      char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTQueryStationStatusConfEvent - Service Response Private Data

typedef struct
{
      ATTEventTypeeventType;// ATT_QUERY_STATION_STATUS_CONF
      union
      {
          ATTQueryStationStatusConfEvent_tqueryStationStatus;
      }u;
      charheap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTQueryStationStatusConfEvent_t
{
      BooleanstationStatus;// TRUE = busy, FALSE = idle
} ATTQueryStationStatusConfEvent_t;
```

Query Time Of Day Service

Summary

- Direction: Client to Switch
- Function: cstaEscapeService()
- Confirmation Event: CSTAEscapeServiceConfEvent
- Private Data Function: attQueryTimeOfDay()
- Private Data Confirmation Event: ATTQueryTimeofDayConfEvent
- Service Parameters: noData
- Private Parameters: noData
- Ack Parameters: noData
- Ack Private Parameters: time
- Nak Parameter: universalFailure

Functional Description:

The Query Time of Day Service provides the switch information for the year, month, day, hour, minute, and second.

Service Parameters:

noData	None for this service.

Ack Parameters:

noData	None for this service.

Ack Private Parameters:

time	[mandatory] Specifies the year, month, day, hour, minute, and second. The year 1999 is specified by two digits - 99. The year 2000 is specified by one digit - 0. The year 2001 is specified by one digit - 1. The year 2002 is specified by one digit - 2, and so forth.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786

Detailed Information:

None

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      PrivateData_t*privateData);


// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAEscapeSvcConfEvent_tescapeService;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t {
      Nulltypenull
} CSTAEscapeSvcConfEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryTimeOfDay() -  Service Request Private Data Setup Function

RetCode_tattQueryTimeOfDay (// no private parameter,
// but must be called
      ATTPrivateData_t*privateData);

typedef struct ATTPrivateData_t {
      char        vendor[32];
      unsigned shortlength;
      char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTQueryTimeofDayConfEvent - Service Response Private Data

typedef struct
{
      ATTEventTypeeventType;// ATT_QUERY_TOD_CONF
      union
      {
          ATTQueryTODConfEvent_tqueryTOD;
      }u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTQueryTODConfEvent_t
{
      shortyear;
      shortmonth;
      shortday;
      shorthour;
      shortminute;
      shortsecond;
} ATTQueryTODConfEvent_t;
```

Query Trunk Group Service

Summary

- Direction: Client to Switch
- Function: cstaEscapeService()
- Confirmation Event: CSTAEscapeServiceConfEvent
- Private Data Function: attQueryTrunkGroup()
- Private Data Confirmation Event: ATTQueryTrunkGroupConfEvent
- Service Parameters: noData
- Private Data Parameters: device
- Ack Parameters: noData
- Ack Private Parameters: idleTrunks, usedTrunks
- Nak Parameter: universalFailure

Functional Description:

The Query Trunk Group Service provides the number of idle trunks and the number of in-use trunks. The sum of the idle and in-use trunks provides the number of trunks in service.

Service Parameters:

noData None for this service.

Private Data Parameters

device [mandatory] Specifies a valid trunk group access code.

Ack Parameters:

noData None for this service.

Ack Private Parameters:

idleTrunks	[mandatory] The number of "idle" trunks in the group.
usedTrunks	[mandatory] The number of "in use" trunks in the group

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device.

Detailed Information:

None

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      PrivateData_t*privateData);


// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAEscapeSvcConfEvent_tescapeService;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t {
      Nulltypenull
} CSTAEscapeSvcConfEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attQueryTrunkGroup() - Service Request Private Data Setup Function

RetCode_tattQueryTrunkGroup (// returns NULL if no
// parameter specified
      ATTPrivateData_t*privateData,
      DeviceID_t *device);

typedef struct ATTPrivateData_t {
      char        vendor[32];
      unsigned shortlength;
      char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

// ATTQueryTrunkGroupConfEvent - Service Response Private Data

typedef struct
{
      ATTEventTypeeventType;// ATT_QUERY_TG_CONF
      union
      {
          ATTQueryTGConfEvent_tqueryTg;
      }u;
char  heap[ATTPRIV_MAX_HEAP];
} ATTEvent_t;

typedef struct ATTQueryTGConfEvent_t
{
      short   idleTrunks;// number of "idle" trunks
// in the group
      short   usedTrunks;// number of "in use" trunks
// in the group
} ATTQueryTGConfEvent_t;
```

Query Universal Call ID Service (Private)

Summary

- Direction: Client to Switch
- Function: cstaEscapeService()
- Confirmation Event: CSTAEscapeServiceConfEvent
- Private Data Function: attQueryUCID()
- Private Data Confirmation Event: ATTQueryUCIDConfEvent
- Service Parameters: noData
- Private Parameters: call
- Ack Parameters: noData
- Ack Private Parameters: ucid
- Nak Parameter: universalFailure

Functional Description:

The Query Universal Call ID Service responds with the Universal Call ID (UCID) for a normal callID. This query may be requested to switch at anytime during the life of a call.

Service Parameters:

noData	None for this service.

Private Parameters:

call	[mandatory] Specifies the normal callID of a call. This is a Connection Identifier. The deviceID is ignored.

Ack Parameters:

noData	None for this service.

Ack Private Parameters:

ucid	[mandatory] Specifies the Universal Call ID (UCID) of the requested call. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros).

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 • VALUE_OUT_OF_RANGE(3) The specified callid value is invalid • OBJECT_NOT_KNOWN(4) The specified callid value is zero • NO_ACTIVE_CALL(24) (CS3/86) An invalid call identifier has been specified in call. • INVALID_FEATURE(15) (CS3/63) The switch software does not support this feature. The switch software release may be earlier than R6.

Detailed Information:

None

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t   cstaEscapeService (
      ACSHandle_tacsHandle,
      InvokeID_tinvokeID,
      PrivateData_t*privateData);


// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;// CSTACONFIRMATION
      EventType_teventType;// CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              InvokeID_tinvokeID;
              union
              {
                  CSTAEscapeSvcConfEvent_tescapeService;
              }u;
          } cstaConfirmation;
      } event;
char  heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t {
      Nulltypenull
} CSTAEscapeSvcConfEvent_t;
```

Chapter 9: Snapshot Service Group

Snapshot Service Group describes services that enable the client to get information about a particular call, or information about calls associated with a particular device. The following sections describes the Snapshot services that Application Enablement Services (AE Services) supports:

Snapshot Call Service

Summary

- Direction: Client to Switch
- Function: *cstaSnapshotCallReq()*
- Confirmation Event: *CSTASnapshotCallConfEvent*
- Private Event: *ATTSnapshotCallConfEvent*
- Service Parameters: *snapshotObj*
- Private Parameters: *DeviceHistory*
- Ack Parameters: *snapshotData*
- Nak Parameter: *universalFailure*

Functional Description:

The Snapshot Call Service provides the following information for each endpoint on the specified call:

- Device ID
- Connection ID
- CSTA Local Connection State

The CSTA Connection state may be one of the following: Unknown, Null, Initiated, Alerting, Queued, Connected, Held, or Failed.

The Device ID may be an on-PBX extension, an alerting extension, or a split hunt group extension (when the call is queued). When a call is queued on more than one split hunt group, only one split hunt group extension is provided in the response to such a query. For calls alerting at various groups (for example, hunt group, TEG, etc.), the group extension is reported to the client application. For calls connected to a member of a group, the group member's extension is reported to the client.

Service Parameters:

snapshotObj	[mandatory] Identifies the call object for which snapshot information is requested. The structure includes the call identifier, the device identifier, and the device type (static or dynamic). Communication Manager ignores the device identifier and device type, so they may have null values.

Private Parameters:

DeviceHistory

The DeviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the DeviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

The DeviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the DeviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the DeviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

Ack Parameters:

snapshotData	[mandatory] Contains all the snapshot information for the call for which the request was made. The structure includes a count of how many device endpoints are on the call as well as the following detailed information for each endpoint: Device ID, Call ID, and Local Connection State of the call at the device.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 INVALID_CSTA_CALL_IDENTIFIER (11) An invalid call identifier has been specified in snapshotObj. INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in snapshotObj.

Syntax

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

// cstaSnapshotCallReq() - Service Request

RetCode_t   cstaSnapshotCallReq
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    ConnectionID_t  *snapshotObj;
    PrivateData_t   *privateData);

// CSTASnapshotCallConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass; // CSTACONFIRMATION
    EventType_t     eventType;  // CSTA_SNAPSHOT_CALL_CONF
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t    eventHeader;
        union
        {
            struct
            {
                InvokeID_t  invokeID;
                union
                {
                    CSTASnapshotCallConfEvent_t snapshotCall;
                }u;

            } cstaConfirmation;
        } event;
        char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTASnapshotCallConfEvent_t {
        CSTASnapshotCallData_t      snapshotCall;
} CSTASnapshotCallConfEvent_t;

typedef struct CSTASnapshotCallData_t {
        int         count; // count of calls
        struct      CSTASnapshotCallResponseInfo_t   *info;
} CSTASnapshotCallData_t;
```

Syntax (Continued)

```
typedef struct CSTASnapshotCallResponseInfo_t {
    SubjectDeviceID_t       deviceOnCall;
    ConnectionID_t          callIdentifier;
    LocalConnectionState_t  localConnectionState;
} CSTASnapshotCallResponseInfo_t;
```

Private Data Version 7 and 8 Syntax

The CSTA Snapshot Call includes a private data event, *ATTSnapshotCallConfEvent* for private data version 7. The ATTSnapshotCallConfEvent uses the deviceHistory private data parameter.

```
typedef struct ATTSnapshotCallConfEvent_t {
    DeviceHistory_t deviceHistory;
} ATTSnapshotCallConfEvent_t;
```

Snapshot Device Service

Summary

- Direction: Client to Switch
- Function: *cstaSnapshotDeviceReq()*
- Confirmation Event: *CSTASnapshotDeviceConfEvent*
- Private Data Confirmation Event: *ATTSnapshotDeviceConfEvent* (private data version 5), *ATTV4SnapshotDeviceConfEvent* (private data versions 2-4)
- Service Parameters: *snapshotObj*
- Ack Parameters: *snapshotDevice*
- Ack Private Parameters: *attSnapshotDevice*
- Nak Parameter: *universalFailure*

Functional Description:

The Snapshot Device Service provides information about calls associated with a given CSTA device. The information identifies each call and indicates the CSTA local connection state for all devices on each call.

Note:
In the Release 2.0 product, the list of connection states for each call may not be a complete list.

Service Parameters:

snapshotObj	[mandatory] Must be a valid device.

Ack Parameters:

snapshotDevice	[mandatory] Contains a sequence of information about each call on the device. Information for each call includes the connectionID and a sequence of local connection states for each connection in the call.

Ack Private Parameters:

attsnapshotDevice	[mandatory] Contains a sequence of information about each call on the device. Information for each call includes the connectionID and the G3 call state for each call at the snapshot device.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 • INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier has been specified in device or forwardingDN.

Detailed Information:

- The ECMA-180 definition for the ack response does not distinguish between the call states for each individual connection making up a call. This is a deficiency because there is no way to correlate the local connection state to a particular connection ID within a call. To overcome this deficiency, Communication Manager always returns the local connection state for the queried device first in the list for each of the calls. The response contains lists of connection states for each call at the snapshot device.
- Information for a maximum of 10 calls is provided for the snapshot device. This is a Communication Manager limit.
- The mapping from the Communication Manager call state to the CSTA local call state (provided in the CSTA response) is as follows:

G3 Local Call State	CSTA Local Call State
Initiate	Initiated
Alerting	Alerting
Connected	Connected
Held	Hold
Bridged	Null
Other	None (CS_NONE)

- The bridged state is a Communication Manager private local connection state that is not defined in the CSTA. This state indicates that a call is present at a bridged, simulated bridged, button TEG, or PCOL appearance, and the call is neither ringing nor connected at the station. The bridged connection state is reported in the private data of a Snapshot Device Confirmation Event and it has a CSTA null (CS_NULL) state. Thus a device with the null state in the Snapshot Device Confirmation Event is bridged.

- A device with the bridged state can join the call by manually answering the call (press the line appearance) or through the cstaAnswerCall service. Once a bridged device is connected to a call, its state becomes connected. After a bridged device becomes connected, it can drop from the call and become bridged again, if the call is not cleared.

Syntax

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

// cstaSnapshotDeviceReq() - Service Request

RetCode_t   cstaSnapshotDeviceReq
    ACSHandle_t     acsHandle,
    InvokeID_t      invokeID,
    DeviceID_t      *snapshotObj;


// CSTASnapshotDeviceReqConfEvent - Service Response

typedef struct
{
    ACSHandle_t     acsHandle;
    EventClass_t    eventClass; // CSTACONFIRMATION
    EventType_t     eventType;  // CSTA_SNAPSHOT_DEVICE_CONF
} ACSEventHeader_t;

typedef struct
{
        ACSEventHeader_t    eventHeader;
        union
        {
            struct
            {
                InvokeID_t  invokeID;
                union
                {
                    CSTASnapshotDeviceConfEvent_t snapshotDevice;
                }u;

            } cstaConfirmation;
        } event;
        char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTASnapshotDeviceConfEvent_t {
        CSTASnapshotDeviceData_t    snapshotData;
} CSTASnapshotDeviceConfEvent_t;

typedef struct CSTASnapshotDeviceData_t {
        int         count; // count of calls on device
        struct      CSTASnapshotDeviceResponseInfo_t     *info;
                                        // info for each call
} CSTASnapshotDeviceData_t;
```

Syntax (Continued)

```
typedef struct CSTASnapshotDeviceResponseInfo_t {
    ConnectionID_t          callIdentifier;
                        // local connection ID
    CSTACallState_t         callstate;
                        // list of connection states
} CSTASnapshotDeviceResponseInfo_t;

typedef struct CSTACallState_t {
    int     count;      // count of connections on call
    LocalConnectionState_t  *state;
                        // list of connection states
} CSTACallState_t;

typedef enum LocalConnectionState_t {
CS_NONE     = -1,   // not an expected snapshot device
                    // response
CS_NULL     = 0,    // indicates a bridged state
CS_INITIATE = 1,
CS_ALERTING = 2,
CS_CONNECT  = 3,
CS_HOLD     = 4,
CS_QUEUED   = 5,
CS_FAIL     = 6,
} LocalConnectionState_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTSnapshotDeviceConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t  eventType; // ATT_SNAPSHOT_DEVICE_CONF
    union
    {
        ATTSnapshotDeviceConfEvent_t    snapshotDevice;
    } u;
} ATTEvent_t;

typedef struct ATTSnapshotDeviceConfEvent_t
{
    int                 count;
    ATTSnapshotDevice_t *pSnapshotDevice;
} ATTSnapshotDeviceConfEvent_t;

typedef struct ATTSnapshotDevice_t
{
    ConnectionID_t          call;
    ATTLocalCallState_t     state;
} ATTSnapshotDevice_t;

typedef enum ATTLocalCallState_t
{
    ATT_CS_INITIATED    = 1,
    ATT_CS_ALERTING     = 2,
    ATT_CS_CONNECTED    = 3,
    ATT_CS_HELD         = 4,
    ATT_CS_BRIDGED      = 5,
    ATT_CS_OTHER        = 6
} ATTLocalCallState_t;
```

Chapter 10: Monitor Service Group

Monitor Service Group, provides an overview of all monitor services and individual descriptions of each service, as follows:

Overview

This overview provides a high level description of each of the monitor services that Avaya Computer Telephony supports. Additionally, it includes the following topics.

Change Monitor Filter Service —cstaChangeMonitorFilter()

This service is used by a client application to change the filter options in a previously requested monitor association.

Monitor Call Service — cstaMonitorCall()

This service provides call event reports passed by the call filter for a single call to an application, but does not provide any agent, feature, or maintenance event reports.

Monitor Calls Via Device Service - cstaMonitorCallsViaDevice()

This service[1] provides call event reports passed by the call filter for all devices on all calls that involve a VDN or an ACD Split device. Event reports are provided for calls that arrive at the device after the monitor request is acknowledged. Events that occurred prior to the monitor request are not reported. If a call is diverted, forwarded, conferenced, or non-monitored ACD or VDN device, subsequent events of that call are reported. Special rules apply to the event reports when the call is diverted, forwarded, conferenced, or transferred. Details are provided in later sections.

This service does not provide any agent, feature, or maintenance event reports.

Monitor Device Service - cstaMonitorDevice()

This service[2] provides call event reports passed by the call filter for all devices on all calls at a station device. Event reports are provided for calls that occurred prior to the monitor request and arrive at the device after the monitor request is acknowledged. If a call is dropped, no further events of the call are reported, forwarded, or transferred from the device, and the device has ceased to participate in the call.

The service also provides feature event reports passed by the filter for a monitored station device as well as agent event reports passed by the filter for a monitored ACD Split device.

The service does not provide maintenance event reports.

Monitor Ended Event - CSTAMonitorEndedEvent

The switch uses this event report to notify a client application that a previously requested Monitor Service has been canceled.

Monitor Stop On Call Service (Private) - attMonitorStopOnCall()

An application uses this service to stop call event reports of a specific call on a monitored device.

1. The Monitor Calls Via Device Service is the call-type Monitor Start Service on a static device identifier in ECMA-179.
2. The Monitor Device Service is the device-type Monitor Start Service on a static device identifier in ECMA-179.

Monitor Stop Service - cstaMonitorStop()

An application uses this service to cancel a previously requested Monitor Service.

Event Filters and Monitor Services

Table 14: Event Filters and Monitor Services shows the relationship between event filters and monitor services.

- An "On" means that this filter is always turned on in the service request confirmation event or the change filter service request confirmation event. This monitor request will never receive this event.
- An "On/Off" means that this filter can be turned on or off in the service request or in the change filter service request and the active filters will be specified in the confirmation event. If a filter is set to on, this monitor request will not receive that event.

Note:
If the Private Filter is set to On, all ATT private event filters (Entered Digits) will be automatically set to On, meaning that there will be no ATT private events for the monitor request.

Table 14: Event Filters and Monitor Services

Event Filters	Monitor Call	Monitor Device (Station)	Monitor Device (ACD Split)	Monitor Device (Trunk or All Trunks)	Monitor Calls Via Device (VDN or ACD Split)
Call Event Filters					
Advice of Charge (private data 5)	On/Off	On/Off	On/Off	On/Off	On/Off
Call Cleared	On/Off	On	On	On	On/Off
Conferenced	On/Off	On/Off	On	On	On/Off
Connection Cleared	On/Off	On/Off	On	On	On/Off
Delivered	On/Off	On/Off	On	On	On/Off
Diverted	On	On/Off	On	On	On/Off
Entered Digits (private)	On/Off	On	On	On	On/Off
Established	On/Off	On/Off	On	On	On/Off

Table 14: Event Filters and Monitor Services (continued)

Event Filters	Monitor Call	Monitor Device (Station)	Monitor Device (ACD Split)	Monitor Device (Trunk or All Trunks)	Monitor Calls Via Device (VDN or ACD Split)
Failed	On/Off	On/Off	On	On	On/Off
Held	On/Off	On/Off	On	On	On/Off
Network Reached	On/Off	On/Off	On	On	On/Off
Originated	On	On/Off[1]	On/Off[1]	On	On
Queued	On/Off	On/Off	On	On	On/Off
Retrieved	On/Off	On/Off	On	On	On/Off
Service Initiated	On	On/Off	On	On	On
Transferred	On/Off	On/Off	On	On	On/Off
Agent Event Filters					
Logged On	On	On/Off[1]	On/Off[1]	On	On
Logged Off	On	On/Off	On/Off	On	On
Not Ready	On	On	On	On	On
Ready	On	On	On	On	On
Work Not Ready	On	On	On	On	On
Work Ready	On	On	On	On	On
Feature Event Filters					
Call Information	On	On	On	On	On
Do Not Disturb	On	On	On	On	On
Forwarding	On	On	On	On	On
Message Waiting	On	On	On	On	On
Maintenance Event Filters					
Back in Service	On	On	On	On	On
Out of Service	On	On	On	On	On
Private Filter	On/Off	On/Off	On/Off	On/Off	On/Off

1. For PBX Version G3V3 and earlier, Originated and Agent Logged On are always filtered (On).

The localConnectionInfo Parameter for Monitor Services

Table 15 shows the availability of the localConnectionInfo parameter for the monitor services. These definitions follow the CSTA specification.

Table 15: localConnectionInfo for monitor services

Parameter	Monitor Call	Monitor Device (Station)	Monitor Device (ACD Split)	Monitor Device (Trunk or All Trunks)	Monitor Calls Via Device (VDN or ACD Split)
localConnectionInfo	not supported	supported	not supported	not supported	not supported

Change Monitor Filter Service

Summary

- Direction: Client to Switch
- Function: *cstaChangeMonitorFilter()*
- Confirmation Event: *CSTAChangeMonitorFilterConfEvent*
- Private Data Function: *attMonitorFilterExt()* (private data version 5), *attMonitorFilter()* (private data versions 2-4)
- Private Data Confirmation Event: *ATTMonitorConfEvent* (private data version 5), *ATTV4MontorConfEvent* (private data versions 2-4)
- Service Parameters: *monitorCrossRefID, filterList*
- Private Parameters: *privateFilter*
- Ack Parameters: *filterList*
- Ack Private Parameters: *usedFilter*
- Nak Parameter: *universalFailure*

Functional Description:

The Change Monitor Filter Service is used by a client application to change the filter options in a previously requested monitor association.

Service Parameters:

monitorCrossRefID	[mandatory] Must be a valid Cross Reference ID that was returned in a previous CSTAMonitorConfEvent of this acsOpenStream session.
filterList	[mandatory — partially supported] Specifies the filters to be changed. Call Filter, Agent Filter, and Private Filter are supported. Setting a filter of an event (for example, CF_CALL_CLEARED=0x8000 is turned on) in the monitorFilter means that the event will be filtered out and no such event reports will be sent to the application. A zero Private Filter means that the application wants to receive the private events. If Private Filter is non-zero, private events will be filtered out. The Feature Filter and Maintenance Filter are not supported. If either is present, it will be ignored.

Private Parameters:

privateFilter [optional] Specifies the Communication Manager private filters to be changed. The following Communication Private Call Filter and Call Event Reports are supported:

- Private data version 5:
  - ATT_ENTERED_DIGITS_FILTER
  - ATT_CHARGE_ADVICE_FILTER
- Private data versions 2-4:
  - ATT_V4_ENTERED_DIGITS_FILTER

See Table 14 to determine which filters are under the control of the application, that is, can be turned on and off.

Ack Parameters:

filterList [optional — partially supported] Specifies the event reports that are to be filtered out on the object being monitored by the application. This may not be the filterList specified in the service request, because filters for events that are not supported by Communication Manager and filters for events that do not apply to the monitored object are always turned on in filterList. All event reports in Maintenance Filter are set to ON, meaning that there are no reports supported for these events.

Ack Private Parameters:

usedFilter [optional] Specifies the G3 Private Event Reports that are to be filtered out on the object being monitored by the application.

Nak Parameter:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786

INVALID_CROSS_REF_ID (17) The service request specified a Cross Reference ID that is not in use at this time.

Detailed Information:

See Event Report Detailed Information on page 677.

Syntax

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

// cstaChangeMonitorFilter() - Service Request

RetCode_t    cstaChangeMonitorFilter (
    ACSHandle_t                acsHandle,
    InvokeID_t                 invokeID,
    CSTAMonitorCrossRefID_t    monitorCrossRefID,
    CSTAMonitorFilter_t        *filterList,
    PrivateData_t              *privateData);

// CSTAChangeMonitorFilterConfEvent - Service Response

typedef struct
{   ACSHandle_t         acsHandle;
    EventClass_t        eventClass;
                        // CSTACONFIRMATION
    EventType_t         eventType;
                        // CSTA_CHANGE_MONITOR_FILTER_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAChangeMonitorFilterConfEvent_t
    changeMonitorFilter;
            } u;
    } cstaConfirmation;
} event;


} CSTAEvent_t;

typedef struct CSTAChangeMonitorFilterConfEvent_t
{
    CSTAMonitorFilter_t     monitorFilter;
}   CSTAChangeMonitorFilterConfEvent_t;
```

Syntax (Continued)

```
typedef unsigned short  CSTACallFilter_t;
#define                     CF_CALL_CLEARED     0x8000
#define                     CF_CONFERENCED      0x4000
#define                     CF_CONNECTION_CLEARED 0x2000
#define                     CF_DELIVERED        0x1000
#define                     CF_DIVERTED         0x0800
#define                     CF_ESTABLISHED      0x0400
#define                     CF_FAILED           0x0200
#define                     CF_HELD             0x0100
#define                     CF_NETWORK_REACHED  0x0080
#define                     CF_ORIGINATED       0x0040
#define                     CF_QUEUED           0x0020
#define                     CF_RETRIEVED        0x0010
#define                     CF_SERVICE_INITIATED0x0008
#define                     CF_TRANSFERRED      0x0004

typedef unsigned char   CSTAFeatureFilter_t;
#define                     FF_CALL_INFORMATION 0x80
#define                     FF_DO_NOT_DISTURB   0x40
#define                     FF_FORWARDING       0x20
#define                     FF_MESSAGE_WAITING  0x10

typedef unsigned char   CSTAAgentFilter_t;

#define                     AF_LOGGED_ON        0x80
#define                     AF_LOGGED_OFF       0x40
#define                     AF_NOT_READY        0x20
#define                     AF_READY            0x10
#define                     AF_WORK_NOT_READY   0x08
                                // not supported
#define                     AF_WORK_READY       0x04

typedef unsigned char   CSTAMaintenanceFilter_t
                                // not supported
#define                     MF_BACK_IN_SERVICE  0x80
#define                     MF_OUT_OF_SERVICE   0x40

typedef struct CSTAMonitorFilter_t {
    CSTACallFilter_t call;
    CSTAFeatureFilter_t feature;
    CSTAAgentFilter_t agent;
    CSTAMaintenanceFilter_t maintenance; // not supported
    long                    privateFilter;
                                // 0 = private events
                            // non-zero = no private events
} CSTAMonitorFilter_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMonitorFilterExt() - Service Request Private Data
                        // Setup Function

RetCode_t    attMonitorFilterExt(
    ATTPrivateData_t    *privateData,
    ATTPrivateFilter_t  privateFilter);

typedef struct ATTPrivateData_t
{
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef unsigned char   ATTPrivateFilter_t;
#define                 ATT_ENTERED_DIGITS_FILTER   0x80
#define                 ATT_CHARGE_ADVICE_FILTER    0x40

// ATTMonitorConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t  eventType;      // ATT_MONITOR_CONF
    union
    {
        ATTMonitorConfEvent_t       monitorStart;
    } u;
} ATTEvent_t;

typedef struct ATTMonitorConfEvent_t
{
        ATTPrivateFilter_t  usedFilter;
} ATTMonitorConfEvent_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMonitorFilter() - Service Request Private Data
                        // Setup Function

RetCode_t    attMonitorFilterExt(
    ATTPrivateData_t        *privateData,
    ATTV4PrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t
{
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef unsigned char   ATTV4PrivateFilter_t;
#define                 ATTV4_ENTERED_DIGITS_FILTER 0x80

// ATTV4MonitorConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t  eventType;      // ATTV4_MONITOR_CONF
    union
    {
        ATTV4MonitorConfEvent_t     v4monitorStart;
    } u;
} ATTEvent_t;

typedef struct ATTV4MonitorConfEvent_t
{
        ATTV4PrivateFilter_tusedFilter;
} ATTV4MonitorConfEvent_t;
```

Monitor Call Service

Summary

- Direction: Client to Switch
- Function: *cstaMonitorCall()*
- Confirmation Event: *CSTAMonitorConfEvent*
- Private Data Function: *attMonitorFilterExt()* (private data version 5), *attMonitorFilter()* (private data versions 2-4)
- Private Data Confirmation Event: *ATTMonitorCallConfEvent* (private data version 5), *ATTV4MonitorCallConfEvent* (private data versions 2-4)
- Service Parameters: *call, monitorFilter*
- Private Parameters: *privateFilter*
- Ack Parameters: *monitorCrossRefID, monitorFilter*
- Ack Private Parameters: *usedFilter, snapshotCall*
- Nak Parameter: *universalFailure*

Functional Description:

This service provides call event reports passed by the call filter for a call (call) already in progress. Event reports are provided after the monitor request is acknowledged. Events that occurred prior to the monitor request are not reported. A call that is being monitored may have a new call identifier assigned to it after a conference or transfer. In this case, event reports continue for that call with the new call identifier.

The event reports are provided for all endpoints directly connected to the Communication Manager server and, in some cases, for endpoints not directly connected to the Communication Manager server that are involved in a monitored call.

A snapshot of the call is provided in the CSTAMonitorConfEvent. The information provided is equivalent to the information provided in a CSTASnapshotCallConfEvent of the monitored call.

Only Call Filter/Call Event Reports and Private Filter are supported. Agent Event Reports, Feature Event Reports and Maintenance Event Reports are not provided.

Service Parameters:

call [mandatory] ConnectionID of the call to be monitored.

monitorFilter [optional - partially supported] Specifies the filters to be used with call. Only Call Filter/Call Event Reports and Private Filter are supported. If a Call Filter is not present, it defaults to no filter, meaning that all Communication Manager CSTA call events will be reported.

Setting a filter of an event (for example, CF_CALL_CLEARED=0x8000 is turned on) in the monitorFilter means that the event will be filtered out and no such event reports will be sent to the application.

A zero Private Filter means that the application wants to receive the private call events. If Private Filter is non-zero, private call events will be filtered out. The Agent Filter, Feature Filter, and Maintenance Filter are not supported. If one of these is present, it will be ignored.

Private Parameters:

privateFilter [optional] Specifies the Communication Manager private filters to be changed. The following G3 Private Call Filter and Call Event Reports are supported:

- Private data version 5:
  - ATT_ENTERED_DIGITS_FILTER
  - ATT_CHARGE_ADVICE_FILTER
- Private data versions 2-4:
  - ATT_V4_ENTERED_DIGITS_FILTER

See Table 14 to determine which filters are under the control of the application, that is, can be turned on and off.

Ack Parameters:

monitorCrossRefID [mandatory] Contains the handle chosen by the TSAPI Service. This handle is a unique value within an acsOpenStream session for the duration of the monitor and is used by the application to correlate subsequent event reports to the monitor request that initiated them. It also allows the correlation of the Monitor Stop to the original cstaMonitorCall request.

monitorFilter [optional — partially supported] Specifies the event reports that are to be filtered out on the object being monitored by the application. This may not be the monitorFilter specified in the service request, because filters for events that are not supported by Communication Manager and filters for events that do not apply to the monitored object are always turned on in monitorFilter. Only Call Filter and Call Event Reports are supported.

All event reports in Agent Filter, Feature Filter, Maintenance Filter, and Private Filter are set to ON, meaning that there are no reports supported for these events.

Ack Private Parameters:

usedFilter [optional] Specifies the G3 Private Filter and Event Reports that are to be filtered out on the object being monitored by the application.

snapshotCall [optional] Provides information about the device identifier, connection, and the CSTA Connection state for up to six (6) endpoints on the call. The Connection state may be one of the following: Unknown, Null, Initiated, Alerting, Queued, Connected, Held, or Failed. The information provided is equivalent to the information provided in a CSTASnapshotCallConfEvent of the monitored call.

Nak Parameters

:

universalFailure If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786

- INVALID_CONNECTION_ID_FOR_ACTIVE_CALL (23) (CS0/100) The call identifier is outside the range of the maximum call identifier value.
- NO_ACTIVE_CALL (24) (CS3/86) The application has sent an invalid call identifier. Call does not exist or has been cleared.
- RESOURCE_BUSY (33) TSAPI Service is busy processing a cstaMonitorCall service request on the same call. Try again.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) (CS0/50) The user has not subscribed for the requested service.
- OBJECT_MONITOR_LIMIT_EXCEEDED (42) (CS3/40) The maximum number of calls being monitored on Communication Manager was exceeded.
- OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44) (CS3/63) The same call may be monitored by another TSAPI Service. The request cannot be executed because the system limit is exceeded for the maximum number of monitors on a call by TSAPI Services.

Detailed Information:

See also Event Report Detailed Information on page 677.

- Monitor Ended Event Report — When the monitored call is ended before a cstaMonitorStop is received to stop the cstaMonitorCall association, a CSTAMonitorEndedEvent will be sent to the application to terminate the cstaMonitorCall association.
- Monitor Stop On Call Service — When the cstaMonitorCall association is stopped by an attMonitorStopOnCall request before a cstaMonitorStop request is received, a CSTAMonitorEndedEvent will be sent to the application to terminate the cstaMonitorCall association.
- Maximum Requests from Multiple TSAPI Services — See the section titled "G3 CSTA System Capacity" in Chapter 3, G3 CSTA Services Overview.
- Multiple Application Requests — Multiple applications can have multiple cstaMonitorCall requests on one object through one TSAPI Service. An application can have more than one cstaMonitorCall request on one object through one TSAPI Service. However, this is not recommended.
- Advice of Charge Event Report (private data v5) — The ATTChargeAdviceEvent is provided, by an outside service, to streams which have enabled Advice of Charge using attSetAdviceOfCharge() and cstaEscapeService(). Typically, an ATTChargeAdviceEvent will arrive from the provider as a call ends, providing the final charge amount. Generally, the final CSTAMonitorEndedEvent (sent for call monitors at the end of a call) is delayed until that final ATTChargeAdviceEvent arrives. When there is a long delay in the arrival of the final ATTChargeAdviceEvent, the CSTAMonitorEndedEvent will be sent to the application and a final ATTChargeAdviceEvent will not be provided.

Syntax

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

// cstaMonitorCall() - Service Request

RetCode_t    cstaMonitorCall (
    ACSHandle_t                 acsHandle,
    InvokeID_t                  invokeID,
    ConnectionID_t              *call,
    CSTAMonitorFilter_t         *monitorFilter, // supports
                                        // call filter only
    PrivateData_t               *privateData);

// CSTAMonitorConfEvent - Service Response

typedef struct
{
    ACSHandle_t         acsHandle;
    EventClass_t        eventClass;
                        // CSTACONFIRMATION
    EventType_t         eventType;
                        // CSTA_MONITOR_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAMonitorConfEvent_t  monitorStart;
            } u;
    } cstaConfirmation;
} event;


} CSTAEvent_t;

typedef struct CSTAMonitorConfEvent_t
{
    CSTAMonitorCrossRefID_t monitorCrossRefID;
    CSTAMonitorFilter_t     monitorFilter;
}   CSTAMonitorConfEvent_t;
```

Syntax (Continued)

```
typedef long       CSTAMonitorCrossRefID_t;

typedef unsigned short  CSTACallFilter_t;
#define                     CF_CALL_CLEARED         0x8000
#define                     CF_CONFERENCED          0x4000
#define                     CF_CONNECTION_CLEARED   0x2000#define
CF_DELIVERED                                        0x1000
#define                     CF_DIVERTED             0x0800
#define                     CF_ESTABLISHED          0x0400
#define                     CF_FAILED               0x0200
#define                     CF_HELD                 0x0100
#define                     CF_NETWORK_REACHED      0x0080
#define                     CF_ORIGINATED           0x0040
#define                     CF_QUEUED               0x0020
#define                     CF_RETRIEVED            0x0010
#define                     CF_SERVICE_INITIATED    0x0008
#define                     CF_TRANSFERRED          0x0004

typedef unsigned char   CSTAFeatureFilter_t;
                                    // not supported
#define                     FF_CALL_INFORMATION     0x80
#define                     FF_DO_NOT_DISTURB       0x40
#define                     FF_FORWARDING           0x20
#define                     FF_MESSAGE_WAITING      0x10

typedef unsigned char   CSTAAgentFilter_t;
#define                     AF_LOGGED_ON            0x80
#define                     AF_LOGGED_OFF           0x40
#define                     AF_NOT_READY            0x20
#define                     AF_READY                0x10
#define                     AF_WORK_NOT_READY       0x08
#define                     AF_WORK_READY           0x04

typedef unsigned char   CSTAMaintenanceFilter_t;
                                    // not supported
#define                     MF_BACK_IN_SERVICE      0x80
#define                     MF_OUT_OF_SERVICE       0x40

typedef struct CSTAMonitorFilter_t {
    CSTACallFilter_t        call;
CSTAFeatureFilter_t         feature;     // not supported
CSTAMaintenanceFilter_t     maintenance; // not supported
    long                    privateFilter;
                            // 0 = report private events
                            // non-zero = no private events
} CSTAAgentFilter_t agent;
CSTAMonitorFilter_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMonitorFilterExt() - Service Request Private Data
                         // Setup Function

RetCode_t    attMonitorFilterExt(
    ATTPrivateData_t      *privateData,
    ATTPrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t
{
    char          vendor[32];
    ushort        length;
    char          data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef unsigned char      ATTPrivateFilter_t;
#define                    ATT_ENTERED_DIGITS_FILTER      0x80
#define                    ATT_CHARGE_ADVICE_FILTER       0x40

// ATTMonitorCallConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t    eventType;        // ATT_MONITOR_CALL_CONF
    union
    {
        ATTMonitorCallConfEvent_t      monitorCallStart;
    } u;
} ATTEvent_t;

typedef struct ATTMonitorCallConfEvent_t
{
    ATTPrivateFilter_t             usedFilter;
    ATTSnapshotCall_t              snapshotCall;
} ATTMonitorCallConfEvent_t;

typedef struct ATTSnapshotCall_t
{
    int                            count; CSTASnapshotCallResponseInfo_t
*pInfo;
} ATTSnapshotCall_t;

typedef struct CSTASnapshotCallResponseInfo_t
{
    SubjectDeviceID_t              deviceOnCall;
    ConnectionID_t                 callIdentifier;
    LocalConnectionState_t         localConnectionState;
} CSTASnapshotCallResponseInfoEvent_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMonitorFilter() - Service Request Private Data
                          // Setup Function

RetCode_t    attMonitorFilter(
    ATTPrivateData_t      *privateData,
    ATTV4PrivateFilter_t  privateFilter);

typedef struct ATTPrivateData_t
{
    char          vendor[32];
    ushort        length;
    char          data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef unsigned char      ATTV4PrivateFilter_t;
#define                    ATT_V4_ENTERED_DIGITS_FILTER   0x80

// ATTV4MonitorCallConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t    eventType;        // ATTV4_MONITOR_CALL_CONF
    union
    {
        ATTV4MonitorCallConfEvent_t    v4monitorCallStart;
    } u;
} ATTEvent_t;

typedef struct ATTV4MonitorCallConfEvent_t
{
    ATTV4PrivateFilter_t           usedFilter;
    ATTV4SnapshotCall_t            snapshotCall;
} ATTV4MonitorCallConfEvent_t;

typedef struct ATTV4SnapshotCall_t
{
    short                          count;
    CSTASnapshotCallResponseInfo_t info[ATT_MAX_PARTIES_ON_CALL];
} ATTV4SnapshotCall_t;

typedef struct CSTASnapshotCallResponseInfo_t
{
    SubjectDeviceID_t              deviceOnCall;
    ConnectionID_t                 callIdentifier;
    LocalConnectionState_t         localConnectionState;
} CSTASnapshotCallResponseInfoEvent_t;
```

Monitor Calls Via Device Service

Summary

- Direction: Client to Switch
- Function: *cstaMonitorCallsViaDevice()*
- Confirmation Event: *CSTAMonitorConfEvent*
- Private Data Function: attMonitorCallsViaDevice() (private data version 7 or later), attMonitorFilterExt() (private data version 5 or later), attMonitorFilter()
- Private Data Confirmation Event: *ATTMonitorConfEvent* (private data version 5 or later), *ATTV4MonitorConfEvent* (private data versions 2-4)
- Service Parameters: *deviceID, monitorFilter*
- Private Parameters: *privateFilter* and *flowPredictiveCallEvents*
- Ack Parameters: *monitorCrossRefID, monitorFilter*
- Ack Private Parameters: *usedFilter*
- Nak Parameter: *universalFailure*

Functional Description:

This service provides call event reports passed by the call filter for all devices on all calls that involve the device (deviceID). Event reports are provided for calls that arrive at the device after the monitor request is acknowledged. Events for calls that occurred prior to the monitor request are not reported. There are feature interactions between two cstaMonitorCallsViaDevice() requests on different monitored ACD or VDN devices.

Note:
There are no feature interactions between a cstaMonitorCallsViaDevice() request and a cstaMonitorDevice request. There are no feature interactions between a cstaMonitorDevice request and another cstaMonitorDevice request.

The event reports are provided for all end points directly connected to the Communication Manager server and may be present for certain types of endpoints not directly connected to the Communication Manager server that are involved in the monitored device.

This service supports only VDN and ACD Split devices, but not station devices. Use cstaMonitorDevice service to monitor stations.

Only Call Filter/Call Event Reports and Private Filter are supported. Agent Event Reports, Feature Event Reports, and Maintenance Event Reports are not supported.

Service Parameters:

deviceID [mandatory] A valid on-PBX VDN or ACD Split extension to be monitored. A station extension is invalid.

monitorFilter [optional — partially supported] Specifies the filters to be used with deviceID. Only Call Filter/Call Event Reports and Private Filter are supported. If a Call Filter is not present, it defaults to no filter, meaning that all Communication Manager CSTA call events will be reported.

Setting a filter of an event (for example, CF_CALL_CLEARED=0x8000 is turned on) in the monitorFilter means that the event will be filtered out and no such event reports will be sent to the application.

A zero Private Filter means that the application wants to receive the private call events. If Private Filter is non-zero, private call events will be filtered out.

The Agent Filter, Feature Filter, and Maintenance Filter are not supported. If one of these is present, it will be ignored.

Private Parameters:

privateFilter [optional] Specifies the Communication Manager private filters to be changed. The following G3 Private Call Filter and Call Event Reports are supported:

- Private data version 5:
  - ATT_ENTERED_DIGITS_FILTER
  - ATT_CHARGE_ADVICE_FILTER
- Private data versions 2-4:
  - ATT_V4_ENTERED_DIGITS_FILTER

See Table 14 to determine which filters are under the control of the application, that is, can be turned on and off.

flowPredictiveCallEvents [optional] Specifies whether "first-leg" Predictive Dial call events should be reported on this monitor.

For a predictive dial call, normally the first event that would be reported on this monitor is a CSTADeliveredEvent indicating that the call arrived at the VDN.

When the application specifies this parameter as FALSE, this behavior is unchanged.

When the application specifies this parameter as TRUE, the monitor also receives events for the outbound call to the calledDevice in the cstaMakePredictiveCall() request. These events may include:

- a CSTADeliveredEvent indicating that the call arrived at the calledDevice;
- a CSTAEstablishedEvent indicating that the call has been answered by the calledDevice; or
- a CSTAConnectionClearedEvent indicating that the connection has been cleared at the calledDevice.

Ack Parameters:

monitorCrossRefID	[mandatory] Contains the handle chosen by the TSAPI Service. This handle is a unique value within an acsOpenStream session for the duration of the monitor and is used by the application to correlate subsequent event reports to the monitor request that initiated them. It also allows the correlation of the Monitor Stop to the original cstaMonitorCallsViaDevice() request.
monitorFilter	[optional — partially supported] Specifies the event reports that are to be filtered out for the object being monitored by the application. This may not be the monitorFilter specified in the service request because filters for events that are not supported by Communication Manager and filters for events that do not apply to the monitored device are always turned on in monitorFilter. Only Call Filter and Call Event Reports are supported. All event reports in Agent Filter, Feature Filter, Maintenance Filter are set to "ON", meaning that there are no reports supported for these events.

Ack Private Parameter:

usedFilter	[optional] Specifies the G3 private event reports that are to be filtered out on the object being monitored by the application.

Nak Parameters

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786. • REQUEST_INCOMPATIBLE_WITH_OBJECT (2) Monitored object is not administered correctly in the switch. The monitored object is an adjunct-controlled split or a vector-controlled split. • INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in deviceID. • RESOURCE_BUSY (33) TSAPI Service is busy processing a cstaMonitorCallsViaDevice() service request on the same device. Try again. • GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) The user has not subscribed to the requested service. • OBJECT_MONITOR_LIMIT_EXCEEDED (42) The request cannot be executed because the system limit would be exceeded for the maximum number of monitors.

Detailed Information:

See also Event Report Detailed Information on page 677.

- ACD split — An ACD split can be monitored by this service only for Call Event Reports.

- Adjunct-Controlled Splits — A cstaMonitorCallsViaDevice() request will be denied (REQUEST_INCOMPATIBLE_WITH_OBJECT) if the monitored object is an adjunct-controlled split.
- Maximum Number of Objects that can be Monitored — See "G3 CSTA System Capacity" section in Chapter 3. G3 CSTA Services Overview.
- Multiple Requests — Multiple applications can have multiple cstaMonitorCallsViaDevice() requests on one object. An application can have more than one cstaMonitorCallsViaDevice() request on one object; however, the latter is not recommended.
- Personal Central Office Line (PCOL) — Members of a PCOL may be monitored. PCOL behaves like bridging for the purpose of event reporting.
- Skill Hunt Groups -- A skill hunt group (split) cannot be monitored directly by an application. The VDN providing access to the vector(s) controlling the hunt group can be monitored instead, if event reports for calls delivered to the hunt group are desired.

Special Rules - Monitor Calls Via Device Service

The following rules apply when a monitored call is diverted, forwarded, or transferred.

- If a call monitored by a cstaMonitorCallsViaDevice() request is diverted to a device that is not monitored by a cstaMonitorCallsViaDevice() request, then there is no Diverted Event Report generated. Subsequent event reports of the call continue.
- If a call monitored by a cstaMonitorCallsViaDevice() at an ACD or VDN device (A) and is diverted to an ACD or VDN device (B) monitored by a cstaMonitorCallsViaDevice() request, then a Diverted Event Report is sent on the monitor for the device (A) that the call left, and no subsequent event reports will be sent for this call on the monitor for device (A). A Delivered Event Report is sent to the monitor for device (B) and subsequent call event reports are sent on the monitor for device (B). The rule is that call event reports of a call are sent to only one cstaMonitorCallsViaDevice() request.
- If a call that is monitored by a cstaMonitorCallsViaDevice() request is merged by a conference/transfer operation with a call that is not monitored by a cstaMonitorCallsViaDevice() request and the resulting call is the one being monitored, a Conferenced/Transferred Event Report is sent to the monitor request and subsequent event reports of the call continue to the same monitor request. If the resulting call is the one not being monitored, a Conferenced/Transferred Event Report with a new callID is sent to the monitor request, a Call Ended Event Report is sent to the monitor request for the abandoned call, and subsequent event reports of the new call continue to be sent to the same request. In this case, the callID for the abandoned call is no longer valid.

- Station — A station cannot be monitored by this service.
- Terminating Extension Group (TEG) — Members of a TEG may be monitored. PCOL behaves like bridging for the purpose of event reporting.

- Vector-Controlled Split — A vector-controlled split cannot be monitored. The VDN providing access to the vector(s) controlling the split should be monitored instead.

Syntax

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

// cstaMonitorCallsViaDevice() - Service Request

RetCode_t    cstaMonitorCallsViaDevice (
    ACSHandle_t                 acsHandle,
    InvokeID_t                  invokeID,
    DeviceID_t                  *deviceID,
                                // must be VDN or ACD split
    CSTAMonitorFilter_t         *monitorFilter, // supports
                                        // call filter only
    PrivateData_t               *privateData);

// CSTAMonitorConfEvent - Service Response

typedef struct
{
    ACSHandle_t         acsHandle;
    EventClass_t        eventClass;
                        // CSTACONFIRMATION
    EventType_t         eventType;
                        // CSTA_MONITOR_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAMonitorConfEvent_t  monitorStart;
            } u;
    } cstaConfirmation;
} event;

} CSTAEvent_t;

typedef struct CSTAMonitorConfEvent_t
{
    CSTAMonitorCrossRefID_t monitorCrossRefID;
    CSTAMonitorFilter_t     monitorFilter;
}   CSTAMonitorConfEvent_t;
```

Syntax (Continued)

```
typedef long       CSTAMonitorCrossRefID_t;

typedef unsigned short  CSTACallFilter_t;
##define                    CF_CONFERENCED          0x4000
#define                     CF_CONNECTION_CLEARED   0x2000

#define                     CF_DELIVERED            0x1000
#define                     CF_DIVERTED             0x0800
#define                     CF_ESTABLISHED          0x0400
#define                     CF_FAILED               0x0200
#define                     CF_HELD                 0x0100
#define                     CF_NETWORK_REACHED      0x0080
#define                     CF_ORIGINATED           0x0040
#define                     CF_QUEUED               0x0020
#define                     CF_RETRIEVED            0x0010
#define                     CF_SERVICE_INITIATED    0x0008
#define                     CF_TRANSFERRED          0x0004

typedef unsigned char   CSTAFeatureFilter_t;
                                // not supported
#define                     FF_CALL_INFORMATION     0x80
#define                     FF_DO_NOT_DISTURB       0x40
#define                     FF_FORWARDING           0x20
#define                     FF_MESSAGE_WAITING      0x10

typedef unsigned char   CSTAAgentFilter_t;
                                // not supported
#define                     AF_LOGGED_ON            0x80
#define                     AF_LOGGED_OFF           0x40
#define                     AF_NOT_READY            0x20
#define                     AF_READY                0x10
#define                     AF_WORK_NOT_READY       0x08
#define                     AF_WORK_READY           0x04

typedef unsigned char   CSTAMaintenanceFilter_t;
                                // not supported
#define                     MF_BACK_IN_SERVICE      0x80
#define                     MF_OUT_OF_SERVICE       0x40

typedef struct CSTAMonitorFilter_t {
    CSTACallFilter_t        call;
CSTAFeatureFilter_t         feature;     // not supported
CSTAMaintenanceFilter_t     maintenance; // not supported
    long                    privateFilter;
                            // 0 = report private events
                            // non-zero = no private events
} CSTAMonitorFilter_t;
```

Private Data Version 7 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>
// attMonitorCallsViaDevice() - Service Request Private Data
//                              formatting function.
RetCode_t attMonitorCallsViaDevice(
    ATTPrivateData_t    *privateData,
    ATTPrivateFilter_t  privateFilter,
    Boolean             flowPredictiveCallEvents);
typedef struct ATTPrivateData_t
{
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;
typedef unsigned char   ATTPrivateFilter_t;
#define                 ATT_ENTERED_DIGITS_FILTER    0x80
#define                 ATT_CHARGE_ADVICE_FILTER     0x40

// ATTMonitorConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t eventType; // ATT_MONITOR_CONF
    union
    {
        ATTMonitorConfEvent_t monitorStart;
    } u;
} ATTEvent_t;

typedef struct ATTMonitorConfEvent_t
{
    ATTPrivateFilter_t usedFilter;
} ATTMonitorConfEvent_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMonitorFilterExt() - Service Request Private Data
                        // Setup Function

RetCode_t    attMonitorFilterExt(
    ATTPrivateData_t    *privateData,
    ATTPrivateFilter_t  privateFilter);

typedef struct ATTPrivateData_t
{
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef unsigned char   ATTPrivateFilter_t;
#define                 ATT_ENTERED_DIGITS_FILTER   0x80
#define                 ATT_CHARGE_ADVICE_FILTER    0x40

// ATTMonitorConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t  eventType;      // ATT_MONITOR_CONF
    union
    {
        ATTMonitorConfEvent_t       monitorStart;
    } u;
} ATTEvent_t;

typedef struct ATTMonitorConfEvent_t
{
    ATTPrivateFilter_t          usedFilter;
} ATTMonitorConfEvent_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMonitorFilter() - Service Request Private Data
                        // Setup Function

RetCode_t    attMonitorFilter(
    ATTPrivateData_t        *privateData,
    ATTV4PrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t
{
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef unsigned char   ATTV4PrivateFilter_t;
#define                 ATT_V4_ENTERED_DIGITS_FILTER 0x80

// ATTV4MonitorConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t  eventType;      // ATTV4_MONITOR_CONF
    union
    {
        ATTV4MonitorConfEvent_t     v4monitorStart;
    } u;
} ATTEvent_t;

typedef struct ATTV4MonitorConfEvent_t
{
    ATTV4PrivateFilter_t        usedFilter;
} ATTV4MonitorConfEvent_t;
```

Monitor Device Service

Summary

- Direction: Client to Switch
- Function: *cstaMonitorDevice()*
- Confirmation Event: *CSTAMonitorConfEvent*
- Private Data Function: *attMonitorFilterExt()* (private data version 5), attMonitorFilter() (private data versions 2-4)
- Private Data Confirmation Event: *ATTMonitorConfEvent* (private data version 5), *ATTV4MonitorConfEvent* (private data versions 2-4)
- Service Parameters: *deviceID, monitorFilter*
- Private Parameters: *privateFilter*
- Ack Parameters: *monitorCrossRefID, monitorFilter*
- Ack Private Parameters: *usedFilter*
- Nak Parameter: *universalFailure*

Functional Description:

This service provides call event reports passed by the call filter for all devices on all calls at a device. Event reports are provided for calls that occurred previous to the monitor request and arrive at the device after the monitor request is acknowledged. Call events are also provided for calls already present at the device. No further events for a call are reported when that call is dropped, forwarded, or transferred, conferenced, or the device ceases to participate in the call.

The Call Cleared Event is never provided for this service. There are no subsequent event reports for a call after a Connection Cleared or a Diverted Event Report has been received for that call on this service. Reporting of the subsequent call event reports after a Transferred Event Report is dependent on whether the call is merged-in or merged-out from the monitored device.

The event reports are provided for all endpoints directly connected to the Communication Manager server and may in certain cases be provided for endpoints not directly connected to the Communication Manager server that are involved in the calls with the monitored device.

This service supports Call Event Reports for station devices as well as Agent Event Reports for ACD Split devices.

Maintenance Event Reports are not supported.

Note:

DEFINITY ECS Release 5 and later software supports the Charge Advice Event feature. To receive Charge Advice Events, an application must first turn the Charge Advice Event feature on using the Set Advice of Charge Service. (For details, see Set Advice of Charge Service (Private Data Version 5 and Later) on page 338.) If the Charge Advice Event feature is turned on, a trunk group monitored by a cstaMonitorDevice, a station monitored by a cstaMonitorDevice, or a call monitored by a cstaMonitorCall will receive Charge Advice Events. However, this will not occur if the Charge Advice Event is filtered out by the privateFilter in the monitor request and its confirmation event.

Service Parameters:

deviceID

[mandatory] A valid on-PBX extension, trunk group, or ACD extension to be monitored. A VDN extension is invalid.

A trunk group number has the format of a 'T' followed by the trunk group number (e.g., T123), or a 'T' followed by a '#' to indicate all trunk groups (i.e., "T#").

- If a single trunk group number is specified, the monitor session will receive the Charge Advice Event for that trunk group only.
- If "T#" is specified, the monitor session will receive Charge Advice Events from all trunk groups.

A trunk group monitoring will receive the Charge Advice Event only. It will not receive any other call events.

monitorFilter

[optional — partially supported] Specifies the filters to be used with deviceID. Call Filter/Event Reports are supported for station device. If a Call Filter is not present, it defaults to no filter, meaning that all G3 CSTA Call Event Reports will be reported.

The Agent Filter is supported for ACD Split devices.

Setting a filter of an event (for example, CF_CALL_CLEARED=0x8000 is turned on) in the monitorFilter means that the event will be filtered out and no such event reports will be sent to the application.

A zero Private Filter means that the application wants to receive the private events. If Private Filter is non-zero, private events will be filtered out.

The Feature Filter and Maintenance Filter are not supported. If a filter that does not apply to the monitored device is present, it will be ignored.

Private Parameters:

privateFilter — [optional] Specifies the Communication Manager private filters to be changed. The following G3 Private Call Filter and Call Event Reports are supported:

- Private data version 5:
  - ATT_ENTERED_DIGITS_FILTER
  - ATT_CHARGE_ADVICE_FILTER
- Private data versions 2-4:
  - ATT_V4_ENTERED_DIGITS_FILTER

See Table 14 to determine which filters are under the control of the application, that is, can be turned on and off.

Ack Parameters:

monitorCrossRefID — [mandatory] Contains the handle chosen by the TSAPI Service. This handle is a unique value within an acsOpenStream session for the duration of the monitor and is used by the application to correlate subsequent event reports to the monitor request that initiated them. It also allows the correlation of the Monitor Stop to the original Monitor Service request.

monitorFilter — [optional — partially supported] Specifies the event reports that are to be filtered out for the object being monitored by the application. This may not be the monitorFilter specified in the service request because filters for events that are not supported by Communication Manager and filters for events that do not apply to the monitored device are always turned on in monitorFilter. Maintenance Filters are set to "ON", meaning that there are no reports supported for these events.

Ack Private Parameter:

usedFilter — [optional] Specifies the G3 private event reports that are to be filtered out on the object being monitored by the application.

Nak Parameters:

universalFailure

If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 .

- INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid device identifier or extension is specified in deviceID.
- RESOURCE_BUSY (33) TSAPI Service is busy processing a cstaMonitorDevice service request on the same device. Try again.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) The user has not subscribed to the requested service. The Domain (Station) Control feature may not be turned on in Communication Manager. This error will also occur if the device is administered as a CTI station and the "CTI Stations" option is not enabled in Communication Manager; if the request is for any other AWOH (non-CTI) station and the "Phantom Calls" option is not enabled in Communication Manager; or if the request is for a SIP endpoint and the Third Party Call Control Type administered for the station is not set to "Avaya".
- OBJECT_MONITOR_LIMIT_EXCEEDED (42) The request cannot be executed because the system limit would be exceeded for the maximum number of monitor.

Detailed Information:

See also Event Report Detailed Information on page 677.

- ACD split — An ACD split can be monitored by this service only for Agent Event Reports.
- Administration Without Hardware (AWOH) — A station administered without hardware may be monitored. However, no event reports will be provided to the application for this station since there will be no activity at such an extension.
- Analog ports — Analog ports equipped with modems can be monitored by the cstaMonitorDevice Service.
- Attendants and Attendant Groups — An attendant group extension or an individual attendant extension number cannot have a Monitor Device Service.
- Feature Access Monitoring — A station will not prohibit users from access to any enabled switch features. A monitored station can access any enabled switch feature.
- Logical Agents — A logical agent's station extension can be monitored. Login IDs are not valid monitor objects.
- Multiple Requests — Multiple applications can have multiple cstaMonitorDevice requests on one object. An application can have more than one cstaMonitorDevice request on one object. However, this is not recommended.
- Personal Central Office Line (PCOL) — Members of a PCOL may be monitored. PCOL behaves like bridging for the purpose of event reporting.
- Skill Hunt Groups — A skill hunt group (split) cannot be monitored directly by an application. The VDN providing access to the vector(s) controlling the hunt group can be monitored instead if event reports for calls delivered to the hunt group are desired.
- Terminating Extension Group (TEG) — Members of a TEG may be monitored. PCOL behaves like bridging for the purpose of event reporting.
- VDN — A VDN cannot be monitored by this service.
- Vector-Controlled Split — A vector-controlled split cannot be monitored. The VDN providing access to the vector(s) controlling the split should be monitored instead.

Syntax

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

// cstaMonitorDevice() - Service Request

RetCode_t    cstaMonitorDevice (
    ACSHandle_t                 acsHandle,
    InvokeID_t                  invokeID,
    DeviceID_t                  *deviceID,
    CSTAMonitorFilter_t         *monitorFilter,
    PrivateData_t               *privateData);

typedef struct
{
    ACSHandle_t         acsHandle;
    EventClass_t        eventClass;
                        // CSTACONFIRMATION
    EventType_t         eventType;
                        // CSTA_MONITOR_CONF
} ACSEventHeader_t;

// CSTAMonitorConf - Event

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAMonitorConfEvent_t  monitorStart;
            } u;
    } cstaConfirmation;
} event;
char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAMonitorConfEvent_t
{
    CSTAMonitorCrossRefID_t monitorCrossRefID;
    CSTAMonitorFilter_t     monitorFilter;
}   CSTAMonitorConfEvent_t;
```

Syntax (Continued)

```
typedef long       CSTAMonitorCrossRefID_t;

typedef unsigned short  CSTACallFilter_t;
#define                     CF_CALL_CLEARED         0x8000
#define                     CF_CONFERENCED          0x4000
#define                     CF_CONNECTION_CLEARED   0x2000#define
CF_DELIVERED                                        0x1000
#define                     CF_DIVERTED             0x0800
#define                     CF_ESTABLISHED          0x0400
#define                     CF_FAILED               0x0200
#define                     CF_HELD                 0x0100
#define                     CF_NETWORK_REACHED      0x0080
#define                     CF_ORIGINATED           0x0040
#define                     CF_QUEUED               0x0020
#define                     CF_RETRIEVED            0x0010
#define                     CF_SERVICE_INITIATED    0x0008
#define                     CF_TRANSFERRED          0x0004

typedef unsigned char   CSTAFeatureFilter_t;
#define                     FF_CALL_INFORMATION     0x80
#define                     FF_DO_NOT_DISTURB       0x40
#define                     FF_FORWARDING           0x20
#define                     FF_MESSAGE_WAITING      0x10

typedef unsigned char   CSTAAgentFilter_t;
#define                     AF_LOGGED_ON            0x80
#define                     AF_LOGGED_OFF           0x40
#define                     AF_NOT_READY            0x20
#define                     AF_READY                0x08
#define                     AF_WORK_READY           0x04

typedef unsigned char   CSTAMaintenanceFilter_t;
#define                     MF_BACK_IN_SERVICE      0x80
#define                     MF_OUT_OF_SERVICE       0x40

typedef struct CSTAMonitorFilter_t {
    CSTACallFilter_t        call;
CSTAFeatureFilter_t         feature;     CSTAMaintenanceFilter_t
maintenance;
    long                    privateFilter;
                            // 0 = report private events
                            // non-zero = no private events
} CSTAMonitorFilter_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMonitorFilterExt() - Service Request Private Data
                        // Setup Function

RetCode_t    attMonitorFilterExt(
    ATTPrivateData_t    *privateData,
    ATTPrivateFilter_t  privateFilter);

typedef struct ATTPrivateData_t
{
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef unsigned char   ATTPrivateFilter_t;
#define                 ATT_ENTERED_DIGITS_FILTER   0x80
#define                 ATT_CHARGE_ADVICE_FILTER    0x40

// ATTMonitorConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t  eventType;      // ATT_MONITOR_CONF
    union
    {
        ATTMonitorConfEvent_t       monitorStart;
    } u;
} ATTEvent_t;

typedef struct ATTMonitorConfEvent_t
{
    ATTPrivateFilter_t          usedFilter;
} ATTMonitorConfEvent_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMonitorFilter() - Service Request Private Data
                        // Setup Function

RetCode_t    attMonitorFilter(
    ATTPrivateData_t        *privateData,
    ATTV4PrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t
{
    char        vendor[32];
    ushort      length;
    char        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef unsigned char   ATTV4PrivateFilter_t;
#define                 ATT_V4_ENTERED_DIGITS_FILTER 0x80

// ATTV4MonitorConfEvent - Service Response Private Data

typedef struct ATTEvent_t
{
    ATTEventType_t  eventType;      // ATTV4_MONITOR_CONF
    union
    {
        ATTV4MonitorConfEvent_t     v4monitorStart;
    } u;
} ATTEvent_t;

typedef struct ATTV4MonitorConfEvent_t
{
    ATTV4PrivateFilter_t        usedFilter;
} ATTV4MonitorConfEvent_t;
```

Monitor Ended Event Report

Summary

- Direction: Switch to Client
- Event: *CSTAMonitorEndedEvent*
- Service Parameters: *monitorCrossRefID*

Functional Description:

Communication Manager uses the Monitor Ended Event Report to cancel a subscription to a previously requested cstaMonitorCall, cstaMonitorDevice or cstaMonitorCallsViaDevice() Service when a monitor object is removed or changed to become an invalid object by switch administration or when the switch can no longer provide the information. Once a Monitor Ended Event Report is generated, event reports cease to be sent to the client application by the switch and the Cross Reference Association that was established by the original service request is terminated.

Service Parameters:

monitorCrossRefID	[mandatory] Must be a valid Cross Reference ID of this acsOpenStream session.
cause	[optional — supported] Specifies the reason for this event. The following Event Causes are explicitly sent from the switch: EC_NETWORK_NOT_OBTAINABLE The previously monitored object is no longer available due to a CTI link failure. EC_RESOURCES_NOT_AVAILABLE The previously monitored object is no longer available or valid due to switch administration changes or communication protocol error.

Detailed Information:

See Event Report Detailed Information on page 677.

Syntax

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

typedef struct
{
    ACSHandle_t         acsHandle;
    EventClass_t        eventClass;
                        // CSTAUNSOLICITED
    EventType_t         eventType;
                        // CSTA_MONITOR_ENDED
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTAMonitorEndedEvent_t monitorEnded;
            } u;
    } cstaUnsolicited;
} event;
char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAMonitorEndedEvent_t
{
    CSTAEventCause_t        cause;
}   CSTAMonitorEndedEvent_t;
```

Monitor Stop On Call Service (Private)

Summary

- Direction: Client to Switch
- Function: *cstaEscapeService()*
- Confirmation Event: *CSTAEscapeServiceConfEvent*
- Private Data Function: *attMonitorStopOnCall()*
- Private Data Confirmation Event: *ATTMonitorStopOnCallConfEvent*
- Private Parameters: *monitorCrossRefID, callID*
- Ack Parameters: *noData*
- Ack Private Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

An application uses the Monitor Stop On Call Service to stop Call Event Reports of a specific call reported by a cstaMonitorCall, cstaMonitorDevice or cstaMonitorCallsViaDevice() Service when it no longer has an interest in that call. Once a Monitor Stop On Call request has been acknowledged, event reports of that call cease to be sent to the client application. The Monitor Cross Reference Association that was established by the original cstaMonitorDevice or cstaMonitorCallsViaDevice() Service request continues.

If this service applies to a cstaMonitorCall association, the association will be terminated by a Monitor Ended Event Report.

Note:
The current release provides this capability for monitors initiated with the cstaMonitorCall service only. It does not work for the other types of monitors.

Private Parameters:

monitorCrossRefID	[mandatory] Must be a valid Cross Reference ID that was returned in a previous CSTAMonitorConfEvent of this acsOpenStream session.
callID	[mandatory] This is the callID of the call whose event reports are to be stopped.

Ack Parameters:

noData	None for this service.

Ack Private Parameters:

noData	None for this service.

Nak Parameters:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 . INVALID_CROSS_REF_ID (17) The service request specified a Cross Reference ID that is not in use at this time. NO_ACTIVE_CALL (24) The application has sent an invalid call identifier. The call does not exist, the call has been cleared, or the call is not being monitored by the monitoring device.

Detailed Information:

See also Event Report Detailed Information on page 677.

- This service will take effect immediately. Event reports to the application for the specified call will cease after this monitor request. The switch continues to process the call at the monitored object. Call processing is not affected by this service.
- This service will not affect Call Event Reports of the specified call on other monitoring associations.

Syntax

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

// cstaEscapeService() - Service Request

RetCode_t    cstaEscapeService (
    ACSHandle_t                 acsHandle,
    InvokeID_t                  invokeID,
    PrivateData_t               *privateData);

// CSTAEscapeServiceConfEvent - Service Response

typedef struct
{
    ACSHandle_t         acsHandle;
    EventClass_t        eventClass;
                        // CSTACONFIRMATION
    EventType_t         eventType;
                        // CSTA_ESCAPE_SERVICE_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAEscapeSvcConfEvent_t escapeService;
            } u;
    } cstaConfirmation;
} event;

} CSTAEvent_t;

typedef struct CSTAEscapeSvcConfEvent_t
{
    Nulltype                null;
}   CSTAEscapeSvcConfEvent_t;
```

Private Parameter Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attMonitorStopOnCall() - Service Request Private Data
                           // Setup Function

RetCode_t     attMonitorStopOnCall(
    ATTPrivateData_t            *privateData,
    CSTAMonitorCrossRefID_t     monitorCrossRefID,
    ConnectionID_t              *call);

// ATTMonitorStopOnCallEvent - Service Response Private Data
```

If private data accompanies CSTAMonitorStopOnCallConfEvent, then the private data would be stored in the location that the application specified as the privateDate parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then CSTAMonitorStopOnCallConfEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATT private data event structure:

typedef struct
{
    ATTEventType            eventType;
                            // ATT_MONITOR_STOP_ON_CALL_CONF
    union
    {
            ATTMonitorStopOnCallConfEvent_t     monitorStop;
    }u;
} ATTEvent_t;

 typedef struct ATTMonitorStopOnCallConfEvent_t {
    Nulltype    null;
} ATTMonitorStopOnCallConfEvent_t;
```

Monitor Stop Service

Summary

- Direction: Client to Switch
- Function: *cstaMonitorStop()*
- Confirmation Event: *CSTAMonitorStopConfEvent*
- Service Parameters: *monitorCrossRefID*
- Ack Parameters: *noData*
- Nak Parameter: *universalFailure*

Functional Description:

An application uses the Monitor Stop Service to cancel a subscription to a previously requested cstaMonitorCall, cstaMonitorDevice, or cstaMonitorCallsViaDevice() Service when it no longer has an interest in continuing a monitor. Once a Monitor Stop request has been acknowledged, event reports cease to be sent to the client application by the switch and the Cross Reference Association that was established by the original service request is terminated.

Private Parameter:

monitorCrossRefID	[mandatory] Must be a valid Cross Reference ID that was returned in a previous CSTAMonitorConfEvent of this acsOpenStream session.

Ack Parameter:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786 . INVALID_CROSS_REF_ID (17) The service request specified a Cross Reference ID that is not in use at this time.

Detailed Information:

See also the Event Report Detailed Information on page 677.

- Switch Operation — This service will take effect immediately. Event reports to the application for calls in progress will stop for this monitor request. The switch continues to process calls at the monitored object. Calls present at the monitored object are not affected by this service.

Syntax

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

RetCode_t    cstaMonitorStop (
    ACSHandle_t                 acsHandle,
    InvokeID_t                  invokeID,
    CSTAMonitorCrossRefID_t     monitorCrossRefID,
    PrivateData_t               *privateData);

typedef struct
{
    ACSHandle_t         acsHandle;
    EventClass_t        eventClass;
                        // CSTACONFIRMATION
    EventType_t         eventType;
                        // CSTA_MONITOR_STOP_CONF
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAMonitorStopConfEvent_t monitorStop;
            } u;
    } cstaConfirmation;
} event;
char heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAMonitorStopConfEvent_t
{
    Nulltype                null;
}   CSTAMonitorStopConfEvent_t;
```

Chapter 11: Event Report Service Group

Event Report Service Group describes the following event messages (or reports) from Avaya Communication Manager to the Application Enablement Services (AE Services) TSAPI Service.

CSTAEventCause and LocalConnectionState

The Event Report Service Group members described in this chapter rely extensively on the CSTAEventCause and LocalConnectionState enumerated types. Figure 6 provides the definition of the CSTAEventCause enumerated type, and Figure 7 provides the definition of the LocalConnectionState enumerated type.

Figure 6: CSTA EventCause definitions

```
typedef enum CSTAEventCause_t {
       EC_NONE = -1,       // no cause value is specified
       EC_ACTIVE_MONITOR = 1,
       EC_ALTERNATE = 2,
       EC_BUSY = 3,
       EC_CALL_BACK = 4,
       EC_CALL_CANCELLED = 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
       EC_NETWORKSIGNAL = 46,
       EC_ALERTTIMEEXPIRED = 60,
       EC_DESTOUTOFORDER = 65,
       EC_NOTSUPPORTEDBEARERSERVICE = 80,
       EC_UNASSIGNED_NUMBER = 81,
       EC_INCOMPATIBLE_BEARER_SERVICE = 87
} CSTAEventCause_t;
```

Figure 7: LocalConnectionState definitions

```
typedef enum LocalConnectionState_t {
       CS_NONE = -1,       // state not known
       CS_NULL = 0,
       CS_INITIATE = 1,
       CS_ALERTING = 2,
       CS_CONNECT = 3,
       CS_HOLD = 4,
       CS_QUEUED = 5,
       CS_FAIL = 6
} LocalConnectionState_t;
```

Certain cause codes will appear in events only if they make sense. See Table 16 for a description of cause code definitions. See Table 17 for a description the cause codes that are possible for each of the call events.

Table 16: Cause Code Definitions

Cause Code	Definition
Active Monitor	an Active Monitor Feature has occurred. This feature typically allows intrusion by a supervisor into an agent call with the ability to speak and listen. The resultant call can be considered as a conference so this cause code may be supplied with the Conferenced Event Report.
Alternate	the call is in the process of being exchanged. This feature is typically found on single-line telephones, where the human interface puts one call on hold and retrieves a held call or answers a waiting call in an atomic action.
Busy	the call encountered a busy tone or device
Call Back	Call Back is a feature invoked (by a user or via CSTA) in an attempt to complete a call that has encountered a busy or no answer condition. As a result of invoking the feature, the failed call is cleared and the call can be considered as queued. The switch may subsequently automatically retry the call (normally when the called party next becomes free). Consequently, this cause code may appear in Event Reports related to the feature invocation (Call Cleared, Connection Cleared and Queued) or related to the subsequent, retried call (Service Initiated, Originated, Delivered, and Established).
Call Canceled	the user has terminated a call without going on-hook.
Call Forward	the call has been redirected via a Call Forwarding feature set for general, unknown, or multiple conditions.
Call Fd. - Immediate	the call has been redirected via a Call Forwarding feature set for all conditions.

Table 16: Cause Code Definitions (continued)

Cause Code	Definition
Call Fd. - Busy	the call has been redirected via a Call Forwarding feature set for a busy endpoint.
Call Fd. - No Answer	the call has been redirected via a Call Forwarding feature set for an endpoint that does not answer.
Call Not Answered	the call was not answered because a timer has elapsed.
Call Pickup	the call has been redirected via a Call Pickup feature.
Camp On	a Camp On feature has been invoked or has matured.
Dest. Not Obtainable	the call could not obtain the destination.
Do Not Disturb	the call encountered a Do Not Disturb condition.
Incompatible Destination	the call encountered an incompatible destination.
Invalid Account Code	the call has an invalid account code.
Key Operation[1]	indicates that the Event Report occurred at a bridged or twin device.
Lockout	the call encountered inter-digit time-out while dialing.
Maintenance	the call encountered a facility or endpoint in a maintenance condition.
Net Congestion	the call encountered a congested network. In some circumstances this cause code indicates that the user is listening to a "No Circuit" Special Information Tone (SIT) from a network that is accompanied by a statement similar to "All circuits are busy..."
Net Not Obtainable	the call could not reach a destination network.
Resources not Available	resources were not available.
Silent Monitor	the event was caused by the invocation of a feature that allows a third party, such as an ACD agent supervisor, to join the call. The joining party can hear the entire conversation, but cannot be heard by either original party. The feature, sometimes called *silent intrusion*, may provide a tone to one or both parties to indicate that they are being monitored. This feature is not the same as a CSTA Monitor request. This cause shall not indicate that a CSTA Monitor has been initiated.
Transfer	a Transfer is in progress or has occurred.
Trunks Busy	the call encountered Trunks Busy.

Table 16: Cause Code Definitions (continued)

Cause Code	Definition
Voice Unit Initiator	indicates that the event was the result of action by automated equipment (voice mail device, voice response unit, announcement) rather than the result of action by a human user.
Network Signal	indicates that the subscriber is absent (no radio signal from cell).
Alert Time Expired	indicates that no user is responding to cell call.
Dest. Out of Order	indicates that the destination is out of order.
Not Supported Bearer Service	indicates that the service/option is not available; unspecified.
Unassigned Number	indicates an unassigned number.
Incompatible Bearer Service	indicates that the bearer capability is not available.

1. Telephone numbers associated primarily with one device often appear also on a second device. One example is a secretary who's phone has mirrored or bridged lines of a supervisor's phone.

Table 17: CSTA Event Report - Cause Relationships

Cause	Call Clr	Conf	Con. Clr	Dlv	Div	Est	Fail	Held	Net. Rch	Orig	Q-ed	Retr	Svc Init.	Tran	Cell Call[1]
Active Monitor		y													
Alternate						y	y	y				y			
Busy							y				y				
Call Back	y		y	y						y	y		y		
Call Canceled	y		y				y						y		
Call Forward				y	y		y	y	y		y				
Call Fd. - Immediate				y	y		y		y		y				
Call Fd. - Busy				y	y		y		y		y				
Call Fd. - No Answer				y	y		y	y	y		y				
Call Not Answered	y		y		y		y								
Call Pickup					y	y									
Camp On				y			y				y				
Dest. not Obtainable			y				y				y				
Do Not Disturb			y		y		y				y				
Incpt. Destination	y		y		y		y								
Invalid Account Code	y						y								
Key Operation	y	y	y	y	y	y	y	y	y	y	y	y	y	y	
Lockout							y								
Maintenance	y						y								
Net Congestion							y				y				
Net Not Obtainable							y				y				
New Call		y		y		y				y				y	
No Available Agents				y	y		y				y				
Overflow	y		y	y	y		y		y		y				
Override	y	y	y	y		y	y			y			y		

Table 17: CSTA Event Report - Cause Relationships (continued)

Cause	Call Clr	Conf	Con. Clr	Dlv	Div	Est	Fail	Held	Net. Rch	Orig	Q-ed	Retr	Svc Init.	Tran	Cell Call[1]
Park			y								y				
Recall		y		y	y	y	y	y			y	y		y	
Redirected				y	y		y		y		y			y	
Reorder Tone							y								
Resrcs. not Available	y		y				y		y		y				
Silent Monitor		y								y					
Transfer				y		y	y	y	y		y	y		y	
Trunks Busy							y				y				
Voice Unit Initiator					y									y	
Network Signal															y
Alert Time Expired															y
Dest. out of Order															y
Not Supported Bearer Service															y
Unassigned Number															y
Incompatible Bearer Service															y

1. CTI cause values for cell phones.

Event Minimization Feature on Communication Manager

If Communication Manager is administered with the **Event Minimization** feature set to **y** for the CTI link connected to the Application Enablement Services TSAPI Service, then only one set of events for a call is sent to the TSAPI Service even if one or more devices are monitored. For example, if a VDN and an agent station are both monitored, only the VDN monitoring will received the Delivered Event.

Note:

The Event Minimization feature must be set to "n" on the switch for the CTI link connected to the Application Enablement Services TSAPI Service.

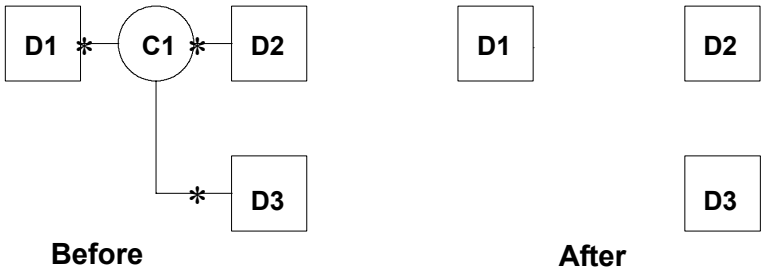
Call Cleared Event

Summary

- Direction: Switch to Client
- Event: *CSTACallClearedEvent*
- Private Data Event: *ATTCallClearedEvent*
- Service Parameters: *monitorCrossRefID*, *clearedCall*, *localConnectionInfo*, *cause*
- Private Parameters: *reason*

Functional Description:

The Call Cleared Event Report indicates that a call is ended. Normally this occurs when the last remaining device or party disconnects from the call. It can also occur when a call is immediately dissolved as the call is conferenced or transferred for a cstaMonitorCallsViaDevice request, but not for a cstaMonitorDevice request.



Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
clearedCall	[mandatory] Specifies the callID of the call that has been cleared. The deviceID is set to 0.
localConnectionInfo	[optional - supported] Always specifies a null state (CS_NULL).
cause	[optional - supported] Specifies a cause when the call is not terminated normally. EC_NONE is specified for normal call termination. ● EC_BUSY - Device busy. ● EC_CALL_CANCELLED - Call rejected or canceled. ● EC_DEST_NOT_OBTAINABLE - Called device is not reachable or wrong number is called. ● EC_CALL_NOT_ANSWERED - Called device not responding or call not answered (maxRings timed out) for a MakePredictiveCall. ● EC_NETWORK_CONGESTION - Network congestion or channel is unacceptable. ● EC_RESOURCES_NOT_AVAILABLE - No circuit or channel is available. ● EC_SINGLE_STEP_TRANSFER (private data version 8 or later) - The call was dissolved as the result of a Single Step Transfer Call operation. (This cause value applies for a Call Cleared event received on a monitor created via cstaMonitorCallsViaDevice(), but not for a monitor created via cstaMonitorDevice().) ● EC_TRANSFER - Call merged due to transfer or conference. ● EC_REORDER_TONE - Intercept SIT treatment - Number changed. ● EC_VOICE_UNIT_INITIATOR - Answer machine is detected for a MakePredictiveCall.

Private Parameters:

reason	[optional] Specifies the reason for this event. The following reason codes are supported: ● AR_NONE- indicate no value specified for reason. ● AR_ANSWER_NORMAL - Answer supervision from the network or internal answer. ● AR_ANSWER_TIMED - Assumed answer based on internal timer. ● AR_ANSWER_VOICE_ENERGY - Voice energy detection from a call classifier. ● AR_ANSWER_MACHINE_DETECTED - Answering machine detected

- AR_SIT_REORDER - Switch equipment congestion
- AR_SIT_NO_CIRCUIT - No circuit or channel available
- AR_SIT_INTERCEPT - Number changed
- AR_SIT_VACANT_CODE - Unassigned number
- AR_SIT_INEFFECTIVE_OTHER - Invalid number
- AR_SIT_UNKNOWN - Normal unspecified

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

// CSTACallClearedEvent

typedef struct
{
      ACSHandle_t     acsHandle;
      EventClass_t    eventClass;     // CSTAUNSOLICITED
      EventType_t     eventType;      // CSTA_CALL_CLEARED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTACallClearedEvent_t  callCleared;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTACallClearedEvent_t
{
      ConnectionID_t       clearedCall;
                          // DeviceID is always 0
      LocalConnectionState_tlocalConnectionInfo;
                          // always CS_NULL
      CSTAEventCause_tcause;
} CSTACallClearedEvent;
```

Private Parameter Syntax

If private data accompanies a CSTACallClearedEvent, then the private data would be stored in the location that the application specified as the private data parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTACallClearedEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTCallClearedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTEvent_t
{
       ATTEventType_teventType;// ATT_CALL_CLEARED
       union
       {
            ATTCallClearedEvent_tcallClearedEvent;
       } u;
} ATTEvent_t;

typedef struct ATTCallClearedEvent_t
{
       ATTReasonCode_treason;
} ATTCallClearedEvent_t;

typedef enum ATTReasonCode_t {
AR_NONE          = 0, // no reason code specified
AR_ANSWER_NORMAL  = 1, // answer supervision from
                       // the network or internal
                       // answer
AR_ANSWER_TIMED   = 2, // assumed answer based on
                       // internal timer
AR_ANSWER_VOICE_ENERGY= 3,  // voice energy detection by
                            // classifier
AR_ANSWER_MACHINE_DETECTED = 4, // answering machine detected
AR_SIT_REORDER    = 5, // switch equipment congestion
AR_SIT_NO_CIRCUIT = 6, // no circuit or channel available
AR_SIT_INTERCEPT  = 7, // number changed
AR_SIT_VACANT_CODE= 8, // unassigned number
AR_SIT_INEFFECTIVE_OTHER = 9,// invalid number
AR_SIT_UNKNOWN    = 10,     // normal unspecified
AR_IN_QUEUE           = 11,// call still in queue - for
                            // Delivered Event only
AR_SERVICE_OBSERVER= 12     // service observer connected
} ATTReasonCode_t
```

Charge Advice Event (Private)

Summary

- Direction: Switch to Client
- Event: *CSTAPrivateStatusEvent*
- Private Data Event: *ATTChargeAdviceEvent*
- Service Parameters: *monitorCrossRefID*
- Private Parameters: *connection, calledDevice, chargingDevice, trunkGroup*, *trunkMember, chargeType, charge, error*

Functional Description:

This event reports the charging units for an outbound call to a trunk group (or all trunk groups) monitoring, a station monitoring, or a call monitoring session. This event is available only if trunk group (or all trunk groups) monitoring is requested to the switch for turning the Charge Advice feature on.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.

Private Parameters:

connection	[mandatory] Specifies the connectionID of the trunk party that generates the charge event. The deviceID is null if split charge is reported due to a conference or transfer.
calledDevice	[mandatory] Specifies the external device that was dialed or requested. This number does not include ARS, FAC, or TAC.
chargingDevice	[mandatory] Specifies the local device that added the trunk group member to the call or an external party if the ISDN-PRI (or R2MFC) calling party number of the caller is available. If no local party is involved, and no calling party is available for an external call, then the TAC of the trunk used on the incoming call will be present. This number indicates to the application the number that may be used at the device that is being charged. Note that this number is not always identical to the CPN or SID that is provided in other event reports reporting on the same call.
trunkGroup	[mandatory] Specifies the trunk group receiving the charge. The number provided correspond to the number used in switch administration, and is not the Trunk Access Code.
trunkMember	[mandatory] Specifies the member of the trunk group receiving the charge.

chargeType [mandatory] Indicates the charge type provide by the network. Valid types are:

- CT_INTERMEDIATE_CHARGE - This is a charge sent by the trunk while the call is active. The charge amounts reported are cumulative. If a call receives two or more consecutive intermediate charges, then the amount from the last intermediate charge replaces the amount(s) of the previous intermediate charges. The amounts are not added to produce a total charge.
- CT_FINAL_CHARGE - This charge is sent by the trunk when a call is dropped. If call CDR outgoing call splitting is not enabled, then the final charge reflects the charge for the entire call.
- CT_SPLIT_CHARGE - CDR outgoing call splitting is used to divide the charge for a call among different users. For example, if an outgoing call is placed by one station and transferred to a second station, and if CDR call splitting is enabled, then CDR and the Charge Advice Events would charge the first station up to the time of the transfer, and the second station after that. A split charge reflects the charge for the call up to the time the split charge is sent (starting at the beginning of the call, or at the previous split charge).
  Any Charge Advice Event received after a split charge will reflect only that portion of the charge that took place after the split charge. If split charges are received for a call, then the total charge for the call can be computed by adding the split charges and the final charge.

charge [mandatory] Specifies the amount of charging units.

error [optional - supported] Indicates a possible error in the charge amount and the reason for the error. It will appear only if there is an error.

- CE_NONE - no error
- CE_NO_FINAL_CHARGE - network failed to provide a final charge for the call (CS3/38)
- CE_LESS_FINAL_CHARGE - final charge provided by the network is less than a previous charge (CS3/38)
- CE_CHARGE_TOO_LARGE - charge provided by the network is too large (CS3/38)
- CE_NETWORK_BUSY - too many calls are waiting for their final charge from the network (CS3/22)

Detailed Information:

- Charge Advice Event Feature - This feature must be turned on via cstaMonitorDevice() with attMonitorDevice().
- Feature Availability - This feature is available starting with DEFINITY G3 Release 5.
- Trunk Group Administration - Only ISDN-PRI trunk groups that have Charge Advice set to "during-on-request" or "automatic" on the switch will receive Charge Advice Events.
- More Than 100 Calls in Call Clearing State - If more than 100 calls are in a call clearing state waiting for charging information, the oldest record will not receive final charge information. In this case a value of 0 and a cause value of CE_NETWORK_BUSY will be reported.

Syntax

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

// CSTAPrivateStatusEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_PRIVATE_STATUS
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      struct
      {   CSTAMonitorCrossRefID_tmonitorCrossRefId;
          union
          {
              CSTAPrivateStatusEvent_tprivateStatus;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef structCSTAPrivateStatusEvent_t
{
      Nulltype        null;

} CSTAPrivateStatusEvent_t;
```

Private Parameter Syntax

If private data accompanies a CSTAPrivateStatusEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAPrivateStatusEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTChargeAdviceEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_t    eventType;    // ATT_CHARGE_ADVICE
      union
      {
          ATTChargeAdviceEvent_t     chargeAdviceEvent;
      } u;
} ATTEvent_t;

typedef struct ATTChargeAdviceEvent_t
{
      ConnectionID_t    connection;
      DeviceID_t        calledDevice;
      DeviceID_t        chargingDevice;
      DeviceID_t        trunkGroup;
      DeviceID_t        trunkMember;
      ATTChargeType_t   chargeType;
      long              charge;
      ATTChargeError_t  error;
} ATTChargeAdviceEvent_t;

typedef enum ATTChargeType_t
{
      CT_INTERMEDIATE_CHARGE     = 1,
      CT_FINAL_CHARGE            = 2,
      CT_SPLIT_CHARGE            = 3
} ATTChargeType_t;

typedef enum ATTChargeError_t
{
      CE_NONE               = 0,
CE_NO_FINAL_CHARGE          = 1,
      CE_LESS_FINAL_CHARGE  = 2,
      CE_CHARGE_TOO_LARGE   = 3,
      CE_NETWORK_BUSY       = 4
} ATTChargeError_t;
```

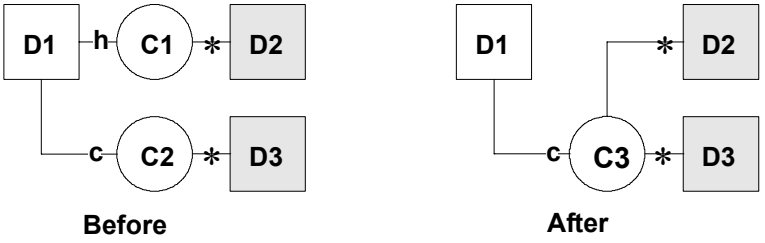
Conferenced Event

Summary

- Direction: Switch to Client
- Event: *CSTAConferencedEvent*
- Private Data Event: *ATTConferencedEvent* (private data version 7), *ATTV6ConferencedEvent* (private data version 6), *ATTV5ConferencedEvent* (private data version 5), *ATTV4ConferencedEvent* (private data version 4), *ATTV3ConferencedEvent* (private data versions 2 and 3)
- Service Parameters: *monitorCrossRefID, primaryOldCall, secondaryOldCall, confController, addedParty, conferenceConnections, localConnectionInfo, cause*
- Private Parameters: *originalCallInfo, distributingDevice, distributingVDN, ucid, trunkList, deviceHistory*

Functional Description:

The Conference Event Report indicates that two calls are conferenced (merged) into one, and no parties are removed from the resulting call in the process. The event may include up to six parties on the resulting call.



The Conferenced Event Report is generated for the following circumstances:

- When an on-PBX station completes a conference by pressing the "conference" button on the voice terminal.
- When an on-PBX station completes a conference after having activated the "supervisor assist" button on the voice set.
- When the on-PBX analog set user flashes the switch hook with one active call and one call on conference and/or transfer hold.
- When an application processor successfully completes a cstaConferenceCall request.
- When the "call park" feature is used in conjunction with the "conference" button on the voice set.

Service Parameters:

monitorCrossRefID	[[mandatory] Contains the handle to the monitor request for which this event is reported.
primaryOldCall	[mandatory] Specifies the callID of the call that was conferenced. This is usually the held call before the conference. This call is ended as a result of the conference.
secondaryOldCall	[mandatory] Specifies the callID of the call that was conferenced. This is usually the active call before the conference. This call was retained by the switch after the conference.
contController	[mandatory] Specifies the device that is controlling the conference. This is the device that set up the conference.
addedParty	[mandatory] Specifies the new conferenced-in device. If the device is an on-PBX station, the extension is specified. If the party is an off-PBX endpoint, then the deviceID is ID_NOT_KNOWN. **Note:** This endpoint's trunk identifier is included in the conferenceConnections list, but not in this parameter. There are call scenarios in which the conference operation joins multiple parties to a call. In such situations, the addedParty will be the extension for the last party to join the call.
conferenceConnections	[optional - supported] Specifies a count of the number of devices and a list of connectionIDs and deviceIDs which resulted from the conference. If a device is on-PBX, the extension is specified. The extension consists of station or group extensions. Group extensions are provided when the conference is to a group and the conference completes before the call is answered by one of the group members (TEG, PCOL, hunt group, or VDN extension). It may contain alerting extensions. The static deviceID of a queued endpoint is set to the split extension of the queue. If a party is off-PBX, then its static device identifier or its previously assigned trunk identifier is specified.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for the cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - limited supported] Specifies the reason for this event. EC_PARK - A call conference was performed for parking a call rather than a true call conference operation. EC_ACTIVE_MONITOR - This is the cause value if the Single Step Conference request is for PT_ACTIVE. For details, see Single Step Conference Call Service (Private Data Version 5 and Later) on page 319 in Chapter 3. EC_SILENT_MONITOR - This is the cause value if the Single Step Conference request is for PT_SILENT. For details, see Single Step Conference Call Service (Private Data Version 5 and Later) on page 319.

Private Parameters:

originalCallInfo

[optional] specifies the original call information. This parameter is sent with this event for the resulting newCall of a cstaConferenceCall request or the retained call of a (manual) conference call operation. The calls being conferenced must be known to the TSAPI Service via the Call Control Services or Monitor Services.

For a cstaConferenceCall, the originalCallInfo includes the original call information originally received by the heldCall specified in the cstaConferenceCall request. For a manual call conference, the originalCallInfo includes the original call information originally received by the primaryOldCall specified in the event report.

The original call information includes:

- ***reason*** - the reason for the originalCallInfo. The following reasons are supported:
  - OR_NONE - no originalCallInfo provided
  - OR_CONFERENCED - call conferenced
- ***callingDevice*** - the original callingDevice received by the heldCall or the primaryOldCall. This parameter is always provided.
- ***calledDevice*** - the original calledDevice received by the heldCall or the primaryOldCall. This parameter is always provided.
- ***trunk*** - the original trunk group received by the heldCall or the primaryOldCall. This parameter is supported by private data versions 2, 3, and 4.
- ***trunkGroup*** - the original trunkGroup received by the heldCall or the primaryOldCall. This parameter is supported by private data version 5 and later only.
- ***trunkMember*** (G3V4 switches and later) - the original trunkMember received by the heldCall or the primaryOldCall.
- ***lookaheadInfo*** - the original lookaheadInfo received by the heldCall or the primaryOldCall.
- ***userEnteredCode*** - the original userEnteredCode received by the heldCall or the primaryOldCall call.
- ***userInfo*** - the original userInfo received by the heldCall or the primaryOldCall call.
- Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo is increased to 96 bytes.
- An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.
- ***ucid*** - the original ucid of the call. This parameter is supported by private data version 5 and later only.

originalCallInfo (continued)

- ***callOriginatorInfo*** - the original callOriginatorInfo received by the activeCall. This parameter is supported by private data version 5 and later only.
- ***flexibleBilling*** - the original flexibleBilling information of the call. This parameter is supported by private data version 5 and later only.
- ***deviceHistory*** - The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. For an explanation of this parameter and the following list of entries, see deviceHistory on page 512
  - **oldDeviceID (M) DeviceID**
  - **EventCause (O) EventCause**
  - **OldConnectionID (O) ConnectionID**

See the Delivered Event section in this chapter for the details of these parameters.

distributingDevice — [optional] Specifies the original distributing device of the call before the call is conferenced. See the Delivered Event section in this chapter for details on the distributingDevice parameter. This parameter is supported by private data version 4 and later

distributingVDN — The VDN extension associated with the distributing device. The field gets set only and exactly under the following conditions.

- When the application monitors the VDN in question and sees the C_OFFERED (translated potentially into a Delivered event, if the application does not filter it out)
- When the application monitors an agent and receives a call that came from that monitored VDN (that is, in the Delivered, Established, Transferred, and Conferenced events).

ucid — [optional] Specifies the Universal Call ID (UCID) of the resulting newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

trunkList — [optional] Specifies a list of up to 5 trunk groups and trunk members. This parameter is supported by private data version 6 and later only. The following options are supported:

- count - The count of the connected parties on the call.
- trunks - An array of 5 trunk group and trunk member IDs, one for each connected party. The following options are supported:
  - connection - The connection ID of one of the parties on the call.
  - trunkGroup - The trunk group of the party referenced by connection.
  - trunkMember - The trunk member of the party referenced by connection.

deviceHistory

The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the deviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

The deviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the deviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the deviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if a ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

The originalCallInfo includes the original call information originally received by the call that is ended (this is usually, but not always, the held call) as the result of the conference.

The following special rules apply:

- If the Conferenced Event was a result of a cstaConferenceCall request, the originalCallInfo and the distributingDevice sent with this Conferenced Event is from the heldCall in the cstaConferenceCall request. Thus the application can control what originalCallInfo and distributingDevice to be sent in a Conferenced Event by putting the original call on hold and specifying it as the heldCall in the cstaConferenceCall request. The primaryOldCall (the call ended as the result of the cstaConferenceCall) is usually the heldCall, but it can be the activeCall.
- If the Conferenced Event was a result of a manual conference, the originalCallInfo and the distributingDevice sent with this Conferenced Event is from the primaryOldCall of the event. Thus the application does not have control of what originalCallInfo and the distributingDevice to be sent in the Conferenced Event. The primaryOldCall (the call ended as the result of the manual conference operation) is usually the held call, but it can be the active call.

Syntax

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

// CSTAConferencedEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_CONFERENCED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTAConferencedEvent_t  conferenced;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAConferencedEvent_t {
      ConnectionID_t          primaryOldCall;
      ConnectionID_t          secondaryOldCall;
      SubjectDeviceID_t       confController;
      SubjectDeviceID_t       addedParty;
      ConnectionList_t        conferenceConnections;
      LocalConnectionState_t  localConnectionInfo;
      CSTAEventCause_t        cause;
} CSTAConferencedEvent_t;

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

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

Private Data Version 7 and 8 Syntax

If private data accompanies a CSTAConferencedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAConferencedEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

The deviceHistory parameter is new for private data version 7.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTConferencedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTConferencedEvent_t {
    ATTOriginalCallInfo_t originalCallInfo;
    CalledDeviceID_t distributingDevice;
    ATTUCID_t       ucid;
    ATTTrunkList_t  trunkList;
    DeviceHistory_t deviceHistory;
    CalledDeviceID_t distributingVDN;
} ATTConferencedEvent_t;
```

Private Data Version 6 Syntax

If private data accompanies a CSTAConferencedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAConferencedEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV6ConferencedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTEvent_t
{
      ATTEventType_t eventType;// ATTV6_CONFERENCED
      union
      {
          ATTV6ConferencedEvent_t v6conferencedEvent;
      } u;
} ATTEvent_t;

typedef struct ATTConferencedEvent_t
{
      ATTOriginalCallInfo_t   originalCallInfo;
      CalledDeviceID_t        distributingDevice;
      ATTUCID_t               ucid;
      ATTTrunkList_t          trunkList;

} ATTConferencedEvent_t;

typedef struct ATTOriginalCallInfo_t
{
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunkGroup;
      DeviceID_t              trunkMember;
      ATTLookaheadInfo_t      lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTUserToUserInfo_t     userInfo;
      ATTUCID_t               ucid;
      ATTCallOriginatorInfo_t callOriginatorInfo;
      Boolean                 flexibleBilling;
} ATTOriginalCallInfo_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef enum ATTReasonForCallInfo_t
{
      OR_NONE               = 0, // indicates not present
      OR_CONSULTATION       = 1,
      OR_CONFERENCED        = 2,
      OR_TRANSFERRED        = 3,
      OR_NEW_CALL           = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_t   CallingDeviceID_t;

typedef ExtendedDeviceID_t   CalledDeviceID_t;

typedef struct ATTLookaheadInfo_t
{
      ATTInterflow_t        type;
      ATTPriority_t         priority;
      short                 hours;
      short                 minutes;
      short                 seconds;
      DeviceID_t            sourceVDN;
      ATTUnicodeDeviceID_t  uSourceVDN;// sourceVDN in Unicode
} ATTLookaheadInfo_t;

typedef enum ATTInterflow_t
{
      LAI_NO_INTERFLOW      = -1,// indicates info not present
      LAI_ALL_INTERFLOW     = 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
      LAI_NOT_IN_QUEUE  = 0,
      LAI_LOW           = 1,
      LAI_MEDIUM        = 2,
      LAI_HIGH          = 3,
      LAI_TOP           = 4
} ATTPriority_t;

typedef unsigned short ATTUnicodeDeviceID_t[64];

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t       type;
      ATTUserEnteredCodeIndicator_t  indicator;
      char                           data[ATT_MAX_USER_CODE];
      DeviceID_t                     collectVDN;
} ATTUserEnteredCode_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE             = -1,// indicates not specified
      UE_ANY              = 0,
      UE_LOGIN_DIGITS     = 2,
      UE_CALL_PROMPTER    = 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR    = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT  = 0,
      UE_ENTERED  = 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTUserToUserInfo_t
{
      ATTUUIProtocolType_ttype;
      struct {
          short           length; // 0 indicates UUI not
                                  // present
          unsigned char   value[ATT_MAX_USER_INFO];
      } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE            = -1,// indicates not specified
      UUI_USER_SPECIFIC   = 0,// user-specific
      UUI_IA5_ASCII       = 4 // null terminated ascii
                              // character string
} ATTUUIProtocolType_t;

typedef charATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t
{
      Boolean hasInfo;// if FALSE, no callOriginatorType
      short       callOriginatorType;
} ATTCallOriginatorInfo_t;
```

Private Data Version 5 Syntax

If private data accompanies a CSTAConferencedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAConferencedEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV5ConferencedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTEvent_t
{
      ATTEventType_teventType;// ATT_CONFERENCED
      union
      {
          ATTV5ConferencedEvent_tconferencedEvent;
      } u;
} ATTEvent_t;

typedef struct ATTV5ConferencedEvent_t
{
      ATTV5OriginalCallInfo_t originalCallInfo;
      CalledDeviceID_t        distributingDevice;
      ATTUCID_t               ucid;
} ATTV5ConferencedEvent_t;

typedef struct ATTV5OriginalCallInfo_t
{
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunkGroup;
      DeviceID_t              trunkMember;
      ATTLookaheadInfo_t      lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTV5UserToUserInfo_t   userInfo;
      ATTUCID_t               ucid;
      ATTV5CallOriginatorInfo_tcallOriginatorInfo;
      Boolean                 flexibleBilling;
} ATTV5OriginalCallInfo_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef enum ATTReasonForCallInfo_t
{
      OR_NONE               = 0, // indicates not present
      OR_CONSULTATION       = 1,
      OR_CONFERENCED        = 2,
      OR_TRANSFERRED        = 3,
      OR_NEW_CALL           = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_t   CallingDeviceID_t;

typedef ExtendedDeviceID_t   CalledDeviceID_t;

typedef struct ATTLookaheadInfo_t
{
      ATTInterflow_t        type;
      ATTPriority_t         priority;
      short                 hours;
      short                 minutes;
      short                 seconds;
      DeviceID_t            sourceVDN;
      ATTUnicodeDeviceID_t  uSourceVDN;// sourceVDN in Unicode
} ATTLookaheadInfo_t;

typedef enum ATTInterflow_t
{
      LAI_NO_INTERFLOW      = -1,// indicates info not present
      LAI_ALL_INTERFLOW     = 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
      LAI_NOT_IN_QUEUE  = 0,
      LAI_LOW           = 1,
      LAI_MEDIUM        = 2,
      LAI_HIGH          = 3,
      LAI_TOP           = 4
} ATTPriority_t;

typedef unsigned short ATTUnicodeDeviceID_t[64];

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t       type;
      ATTUserEnteredCodeIndicator_t  indicator;
      char                           data[33];
      DeviceID_t                     collectVDN;
} ATTUserEnteredCode_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE             = -1,// indicates not specified
      UE_ANY              = 0,
      UE_LOGIN_DIGITS     = 2,
      UE_CALL_PROMPTER    = 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR    = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT  = 0,
      UE_ENTERED  = 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_ttype;
      struct {
          short           length; // 0 indicates UUI not
                                  // present
          unsigned char   value[33];
      } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE            = -1,// indicates not specified
      UUI_USER_SPECIFIC   = 0,// user-specific
      UUI_IA5_ASCII       = 4 // null terminated ascii
                              // character string
} ATTUUIProtocolType_t;

typedef charATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t
{
      Boolean hasInfo;// if FALSE, no callOriginatorType
      short       callOriginatorType;
} ATTCallOriginatorInfo_t;
```

Private Data Version 4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV4ConferencedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTEvent_t
{
      ATTEventType_teventType;// ATTV4_CONFERENCED
      union
      {
          ATTV4ConferencedEvent_tv4conferencedEvent;
      } u;
} ATTEvent_t;

typedef struct ATTV4ConferencedEvent_t
{
      ATTV4OriginalCallInfo_t originalCallInfo;
      CalledDeviceID_t        distributingDevice;
} ATTV4ConferencedEvent_t;

typedef struct ATTV4OriginalCallInfo_t
{
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunk;
      DeviceID_t              trunkMember;
      ATTV4LookaheadInfo_t    lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTV5UserToUserInfo_t   userInfo;
} ATTV4OriginalCallInfo_t;

typedef enum ATTReasonForCallInfo_t
{
      OR_NONE = 0,    // indicates not present
      OR_CONSULTATION     = 1,
      OR_CONFERENCED      = 2,
      OR_TRANSFERRED      = 3,
      OR_NEW_CALL         = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;
```

Private Data Version 4 Syntax (Continued)

```
typedef struct ATTV4LookaheadInfo_t
{
      ATTInterflow_t   type;
      ATTPriority_t    priority;
      short            hours;
      short            minutes;
      short            seconds;
      DeviceID_t       sourceVDN;
} ATTV4LookaheadInfo_t;

typedef enum ATTInterflow_t
{
      LAI_NO_INTERFLOW = -1,    // indicates info not present
      LAI_ALL_INTERFLOW= 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
      LAI_NOT_IN_QUEUE     = 0,
      LAI_LOW              = 1,
      LAI_MEDIUM           = 2,
      LAI_HIGH             = 3,
      LAI_TOP              = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t      type;
      ATTUserEnteredCodeIndicator_tindicator;
      char                          data[33];
      DeviceID_t                    collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE          = -1,    // indicates not specified
      UE_ANY           = 0,
      UE_LOGIN_DIGITS  = 2,
      UE_CALL_PROMPTER = 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT   = 0,
      UE_ENTERED   = 1
} ATTUserEnteredCodeIndicator_t;
```

Private Data Version 4 Syntax (Continued)

```
typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_t type;
      struct {
          short           length; // 0 indicates UUI
                                  // not present
          unsigned charvalue[33];
      } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE            = -1,// indicates not specified
      UUI_USER_SPECIFIC   = 0,// user-specific
      UUI_IA5_ASCII       = 4 // null terminated ascii
                              // character string
} ATTUUIProtocolType_t;
```

Private Data Versions 2 and 3 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV3ConferencedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTEvent_t
{
      ATTEventType_teventType;// ATTV3_CONFERENCED
      union
      {
          ATTV3ConferencedEvent_tv3conferencedEvent;
      } u;
} ATTEvent_t;

typedef struct ATTV3ConferencedEvent_t
{
      ATTV4OriginalCallInfo_toriginalCallInfo;
} ATTV3ConferencedEvent_t;

typedef struct ATTV4OriginalCallInfo_t
{
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunk;
      DeviceID_t              trunkMember;
      ATTV4LookaheadInfo_t    lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTV5UserToUserInfo_t   userInfo;
} ATTV4OriginalCallInfo_t;

typedef enum ATTReasonForCallInfo_t
{
      OR_NONE             = 0,// indicates not present
      OR_CONSULTATION     = 1,
      OR_CONFERENCED      = 2,
      OR_TRANSFERRED      = 3,
      OR_NEW_CALL         = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;
```

Private Data Versions 2 and 3 Syntax (Continued)

```
typedef struct ATTV4LookaheadInfo_t
{
      ATTInterflow_t   type;
      ATTPriority_t    priority;
      short            hours;
      short            minutes;
      short            seconds;
      DeviceID_t       sourceVDN;
} ATTV4LookaheadInfo_t;

typedef enum ATTInterflow_t
{
      LAI_NO_INTERFLOW= -1,// indicates info not present
      LAI_ALL_INTERFLOW= 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
      LAI_NOT_IN_QUEUE = 0,
      LAI_LOW          = 1,
      LAI_MEDIUM       = 2,
      LAI_HIGH         = 3,
      LAI_TOP          = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t      type;
      ATTUserEnteredCodeIndicator_tindicator;
      char                          data[33];
      DeviceID_t                    collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE          = -1,    // indicates not specified
      UE_ANY           = 0,
      UE_LOGIN_DIGITS  = 2,
      UE_CALL_PROMPTER = 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT   = 0,
      UE_ENTERED   = 1
} ATTUserEnteredCodeIndicator_t;
```

Private Data Versions 2 and 3 Syntax (Continued)

```
typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_t type;
      struct {
          short           length; // 0 indicates UUI
                                  // not present
          unsigned char   value[33];
      } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE            = -1,// indicates not specified
      UUI_USER_SPECIFIC   = 0,// user-specific
      UUI_IA5_ASCII       = 4 // null terminated ascii
                              // character string
} ATTUUIProtocolType_t;
```

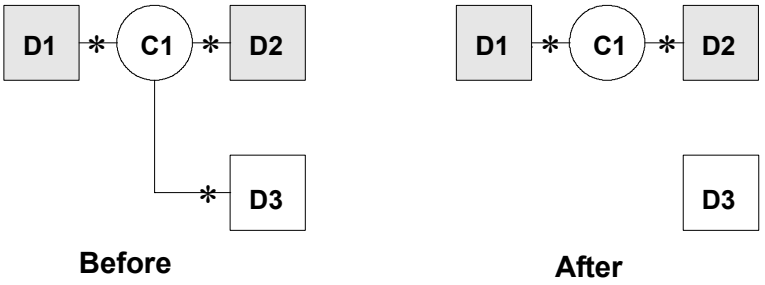
Connection Cleared Event

Summary

- Direction: Switch to Client
- Event: *CSTAConnectionClearedEvent*
- Private Data Event: *ATTConnectionClearedEvent* (private data version 7), *ATTV6ConnectionClearedEvent* (private data version 6), *ATTV5ConnectionClearedEvent* (private data version 2, 3, 4 and 5)
- Service Parameters: *monitorCrossRefID, droppedConnection, releasingDevice, localConnectionInfo, cause*
- Private Parameters: *userInfo, deviceHistory*

Functional Description:

The Connection Cleared Event Report indicates that a device in a call disconnects or is dropped. It does not indicate that a transferring device has left a call in the act of transferring that call.



A Connection Cleared Event Report is generated in the following cases:

- A simulated bridged appearance is dropped when one member drops.
- When an on-PBX party drops from a call.
- When an off-PBX party drops and the ISDN-PRI receives a disconnect message.
- When an off-PBX party drops and the non-ISDN-PRI trunk detects a drop.

A Connection Cleared Event Report is not generated in the following cases:

- A party drops as a result of a transfer operation.
- A split or vector announcement drops.
- Attendant drops a call, if the call was received through the attendant group (0).

- A cstaMakePredictiveCall call is dropped during the call classification stage. (A Call Cleared Event Report is generated instead.)
- A call is delivered to an agent and de-queued from multiple splits as part of vector processing.

This event report is not generated for the last disconnected party on a call for a cstaMonitorCallsViaDevice request. In this case, a Call Cleared Event Report is generated instead. This event is the last event of a call for a cstaMonitorDevice request.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
droppedConnection	[mandatory] Specifies the connection that has been dropped from the call.
releasingDevice	[mandatory] Specifies the dropped device. If the device is on-PBX, then the extension is specified (primary extension for TEGs, PCOLs, bridging). If a party is off-PBX, then its static device identifier or its previously assigned trunk identifier is specified.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - supported] Specifies a cause when the call is not terminated normally. EC_NONE is specified for normal call termination. ● EC_BUSY - Device busy. ● EC_CALL_CANCELLED - Call rejected or canceled. ● EC_DEST_NOT_OBTAINABLE - Called device is not reachable or wrong number is called ● EC_CALL_NOT_ANSWERED - Called device not responding or call not answered (maxRings has timed out) for a MakePredictiveCall. ● EC_NETWORK_CONGESTION - Network congestion or channel is unacceptable. ● EC_RESOURCES_NOT_AVAILABLE - No circuit or channel is available. ● EC_TRANSFER - Call merged due to transfer or conference. ● EC_REORDER_TONE - Intercept SIT treatment - Number changed. ● EC_VOICE_UNIT_INITIATOR - Answer machine is detected for a MakePredictiveCall.

Private Parameters:

userInfo

[optional] Contains user-to-user information. This parameter allows an application to associate caller information, up to 32 or 96 bytes, with a call. This information may be a customer number, credit card number, alphanumeric digits, or a binary string. It is propagated with the call when the call is dropped by a cstaClearConnection with a userInfo and passed to an application in the Connection Cleared Event Report.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo is increased to 96 bytes.

An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE - There is no data provided in the data parameter.
- UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

deviceHistory

The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the deviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

deviceHistory (continued)

The deviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the deviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the deviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if a ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

// CSTAConnectionClearedEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_CONNECTION_CLEARED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
         struct
         {
          CSTAMonitorCrossRefID_tmonitorCrossRefId;
          union
          {
             CSTAConnectionClearedEvent_t  connectionCleared;
          } u;
         } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAConnectionClearedEvent_t
{
      ConnectionID_t      droppedConnection;
      SubjectDeviceID_t   releasingDevice;
      SubjectDeviceID_t   localConnectionInfo;
      CSTAEventCause_t    cause;
} CSTAConnectionClearedEvent_t;
```

Private Data Version 7 and 8 Syntax

If private data accompanies a CSTAConnectionClearedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAConnectionClearedEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

The deviceHistory parameter is new for private data version 7.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTConnectionClearedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTConnectionClearedEvent_t {
    ATTUserToUserInfo_t userInfo;
    DeviceHistory_t deviceHistory;
} ATTConnectionClearedEvent_t;
```

Private Data Version 6 Syntax

If private data accompanies a CSTAConnectionClearedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAConnectionClearedEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV6ConnectionClearedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_teventType;// ATTV6_CONNECTION_CLEARED
      union
      {
          ATTV6ConnectionClearedEvent_t v6connectionCleared;
      } u;
} ATTEvent_t;

typedef struct ATTConnectionClearedEvent_t
{
      ATTUserToUserInfo_tuserInfo;
} ATTConnectionClearedEvent_t;

typedef structATTUserToUserInfo_t  {
      ATTUUIProtocolType_t type;
      struct {
        short           length;// 0 indicates UUI not present
        unsigned char   value[ATT_MAX_USER_INFO];
      } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE            = -1,// indicates not specified
      UUI_USER_SPECIFIC   = 0,// user-specific
      UUI_IA5_ASCII       = 4 // null terminated ascii
// character string
} ATTUUIProtocolType_t
```

Private Data Version 2-5 Syntax

If private data accompanies a CSTAConnectionClearedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAConnectionClearedEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV5ConnectionClearedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_teventType;// ATTV5_CONNECTION_CLEARED
      union
      {
          ATTV5ConnectionClearedEvent_tconnectionCleared;
      } u;
} ATTEvent_t;

typedef struct ATTV5ConnectionClearedEvent_t
{
      ATTV5UserToUserInfo_tuserInfo;
} ATTV5ConnectionClearedEvent_t;

typedef structATTV5UserToUserInfo_t  {
      ATTUUIProtocolType_t type;
      struct {
        short           length;// 0 indicates UUI not present
        unsigned char   value[33];
      } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE            = -1,// indicates not specified
      UUI_USER_SPECIFIC   = 0,// user-specific
      UUI_IA5_ASCII       = 4 // null terminated ascii
// character string
} ATTUUIProtocolType_t
```

Delivered Event

Summary

- Direction: Switch to Client
- Event: *CSTADeliveredEvent*
- Private Data Event: *ATTDeliveredEvent* (private data version 7), *ATTV6DeliveredEvent* (private data version 6), *ATTV5DeliveredEvent* (private data version 5), *ATTV4DeliveredEvent* (private data version 4), *ATTV3DeliveredEvent* (private data versions 2 and 3)
- Service Parameters: *monitorCrossRefID, connection, alertingDevice, callingDevice, calledDevice, lastRedirectionDevice, localConnectionInfo, cause*
- Private Parameters: *deliveredType, trunk, trunkGroup, trunkMember, split, lookaheadInfo, userEnteredCode, userInfo, reason, originalCallInfo, distributingDevice, distributingVDN, ucid, callOriginatorInfo, flexibleBilling, deviceHistory*

Functional Description:

Communication Manager reports two types of Delivered Event Reports

- call delivered to station
- call delivered to ACD/VDN

The type of the Delivered Event is specified in the ATTDeliveredEvent.

Call Delivered to a Station Device

A Delivered Event Report of this type indicates that "alerting" (tone, ring, etc.) is applied to a device or when the switch detects that "alerting" has been applied to a device.

Consecutive Delivered Event Reports are possible. Multiple Delivered Event Reports for multiple devices are also possible (e.g., a principal and its bridging users). The Delivered Event Report is not guaranteed for each call. The Delivered Event Report is not sent for calls that connect to announcements as a result of ACD split forced announcement or announcement vector commands.

The switch generates the Delivered Event Report when the following events occur.

- "Alerting" (tone, ring, etc.) is applied to a device or when the switch detects that "alerting" has been applied to a device.
- The originator of a cstaMakePredictiveCall call is an on-PBX station and ringing or zip tone is started.
- When a call is redirected to an off-PBX station and the ISDN ALERTing message is received from an ISDN-PRI facility.
- When a cstaMakePredictiveCall call is trying to reach an off-PBX station and the call classifier detects precise, imprecise, or special ringing.
- When a cstaMakeCall (or a cstaMakePredictiveCall) call is placed to an off-PBX station, and the ALERTing message is received from the ISDN-PRI facility.

When both a classifier and an ISDN-PRI facility report alerting on a call made by a cstaMakePredictiveCall request, then the first occurrence generates a Delivered Event Report; succeeding reports are not reported by the switch.

Consecutive Delivered Event Reports are possible in the following cases:

- A station is alerted first and the call goes to coverage: a Delivered Event Report is generated each time a new station is alerted.
- A principal and its bridging users are alerted: a Delivered Event Report is generated for the principal and for each bridged station alerted.
- A call is alerting a Terminating Extension Group (TEG); one report is sent for each TEG member alerted.
- A call is alerting a Personal Central Office Line (PCOL); one report is sent for each PCOL member is alerted.
- A call is alerting a coverage/answer point; one report is sent for each alerting member of the coverage answer group.
- A call is alerting a principal with SAC active; one report is sent for the principal and one or more are sent for the coverage points.

Call Delivered to an ACD Device

An ACD device can distribute calls within a switch. If an ACD device is called, normally the call will pass through the device, as the ACD call processing progresses, and eventually be delivered to a station device. Therefore, a call delivered to an ACD device will have multiple Delivered Event Reports before it connects.

There are two types of G3 devices that distribute calls, VDN and ACD split.

A Delivered Event Report of this type is generated when a call is delivered to an ACD device.

- Call Delivered to a VDN - This event is generated when a call is delivered to a monitored VDN.
- Call Delivered to an ACD Split - This event is generated when a call is delivered to a monitored ACD split. The report will be sent even if the ACD split is in night service or has call forwarding active.

A report will be generated for each cstaMonitorCallsViaDevice request that monitors an ACD device through which the call passes.

The Delivered Event Report is not sent for calls that connected to announcements as a result of ACD split forced announcement or announcement vector commands.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
connection	[mandatory] Specifies the endpoint that is alerting.
alertingDevice	[mandatory] Specifies the device that is alerting. If the device being alerted is on-PBX, then the extension of the device is specified (primary extension for TEGs, PCOLs, bridging). If a party is off-PBX, then its static device identifier or its assigned trunk identifier is specified. If the call was delivered to a VDN or ACD split, the monitored object is specified.
callingDevice	[mandatory] Specifies the calling device. The following rules apply: ● For internal calls - the originator’s extension. ● For outgoing calls over PRI facilities - "calling number" from the ISDN SETUP message or its assigned trunk identifier is specified. If the "calling number" does not exist, it is NULL. **Note:** For outgoing calls over non-PRI facilities, there is no Delivered Event Report. A Network Reached Event Report is generated instead. ● For incoming calls over PRI facilities - "calling number" from the ISDN SETUP message or its assigned trunk identifier is specified. If the "calling number" does not exist, it is NULL. ● For incoming calls over non-PRI facilities - the calling party number is generally not available. The assigned trunk identifier is provided instead. **Note:** The trunk identifier is a dynamic device identifier and it can not be used to access a trunk in Communication Manager. ● The trunk identifier is specified only when the calling party number is not available.

callingDevice (continued)

- For calls originated at a bridged call appearance - the principal's extension is specified.
- There is a special case of a cstaMakePredictiveCall call being delivered to a split: in this case, the callingDevice contains the original digits (from the cstaMakePredictiveCall request) provided in the destination field.

calledDevice

[mandatory] Specifies the originally called device. The following rules apply:

- For outgoing calls over PRI facilities - "called number" from the ISDN SETUP message is specified. If the "called number" does not exist (it is NULL), the deviceIDStatus is ID_NOT_KNOWN.
- For outgoing calls over non-PRI facilities - the deviceIDStatus is ID_NOT_KNOWN.
- For incoming calls over PRI facilities - "called number" from the ISDN SETUP message is specified.
- For incoming calls over non-PRI facilities - the principal extension is specified. It may be a group extension for TEG, hunt group, VDN. If the switch is administered to modify the DNIS digits, then the modified DNIS string is specified.
- For incoming calls to PCOL, the deviceIDStatus is ID_NOT_KNOWN.
- For incoming calls to a TEG (principal) group, the TEG group extension is specified.
- For incoming calls to a principal with bridges, the principal's extension is specified.
- If the called device is on-PBX and the call did not come over a PRI facility, the extension of the party dialed is specified.

lastRedirectionDevice

[optional - limited support] Specifies the previous redirection/alerted device in the case where the call was redirected/diverted to the alertingDevice.

localConnectionInfo

[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for cstaMonitorDevice requests only. A value of CS_NONE means the local connection state is unknown.

cause

[optional - supported] Specifies the cause for this event. The following causes are supported:

The following four causes are only available on Communication Manager with G3V4 or later software: EC_CALL_FORWARD, EC_CALL_FORWARD_ALWAYS, EC_CALL_FORWARD_BUSY, EC_CALL_FORWARD_NO_ANSWER.

They have higher precedence than the following three causes: EC_KEY_CONFERENCE, EC_NEW_CALL, EC_REDIRECTED.

For example, if two causes apply to an event; one from the group with higher precedence (for example, EC_CALL_FORWARD_ALWAYS) and one from the group with a lower precedence (for example, EC_NEW_CALL), only the cause from the group with the higher precedence will apply.

cause
(continued)

- EC_CALL_FORWARD (G3V4 or later) - The call has been redirected via one of the following features:
  - Send All Calls
  - Cover All Calls
  - Go to Cover active
  - cstaDeflectCall
- EC_CALL_FORWARD_ALWAYS (G3V4 or later) - The call has been redirected via the Call Forwarding feature.
- EC_CALL_FORWARD_BUSY (G3V4 or later) - The call has been redirected for one of the following reasons:
  - Cover - principal busy
  - Cover - all call appearance busy
- EC_CALL_FORWARD_NO_ANSWER (G3V4 or later) - The call has been redirected because no answer from cover
- EC_KEY_CONFERENCE - Indicates that the event report occurred at a bridged device. This cause has higher precedence than the following two causes.
- EC_NEW_CALL - The call has not yet been redirected.
- EC_REDIRECTED - The call has been redirected.

Private Parameters:

deliveredType	[optional] Specifies the type of the Delivered Event: DELIVERED_TO_ACD - This type indicates that the call is delivered to an ACD split or a VDN device and subsequent Delivered or other events (e.g., QUEUED) may be expected. DELIVERED_TO_STATION - This type indicates that the call is delivered to a station. DELIVERED_OTHER - This type is not in use.
trunkGroup	[optional] Specifies the trunk group number from which the call originated. Beginning with G3V8, trunk group number is provided regardless of whether the callingDevice is available. Prior to G3V8, trunk group number is provided only if the callingDevice is unavailable. This parameter is supported by private data version 5 and later only.
trunk	[optional] Specifies the trunk group number from which the call originated. Trunk group number is provided only if the callingDevice is unavailable. This parameter is supported by private data versions 2, 3, and 4 only.
trunkMember	[optional - limited supported] This parameter is supported beginning with G3V4. It specifies the trunk member number from which the call originated. Beginning with G3V8, trunk member number is provided regardless of whether the callingDevice is available. Prior to G3V8, trunk member number is provided only if the callingDevice is unavailable.
split	[optional] Specifies the ACD split extension to which the call is delivered. This parameter applies to DELIVERED_TO_STATION only.
lookaheadinfo	[optional] Specifies the lookahead interflow information received from the delivered call. Lookahead interflow is a Communication Manager feature that routes some of the incoming calls from one switch to another so that they can be handled more efficiently and will not be lost. The switch that overflows the call provides the lookahead interflow information. A routing application may use the lookahead interflow information to determine the destination of the call. If the lookahead interflow type is set to "LAI_NO_INTERFLOW", no lookahead interflow private data is provided with this event.
userEnteredCode	[optional] Specifies the code/digits that may have been entered by the caller through the G3 call prompting feature or the collected digits feature. If the userEnteredCode code is set to "UE_NONE", no userEnteredCode private data is provided with this event. See the Detailed Information: section for how to setup the switch and application for collecting userEnteredCode.
userInfo	[optional] Contains user-to-user information. This parameter allows an application to associate caller information, up to 32 or 96 bytes, with a call. This information may be a customer number, credit card number, alphanumeric digits, or a binary string. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo is increased to 96 bytes. An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

userInfo (continued)

The following UUI protocol types are supported:

UUI_NONE - There is no data provided in the data parameter.

UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.

UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

reason

[optional] Specifies the reason of this event. The following reason is supported:

- AR_NONE- indicate no value specified for reason.
- AR_IN_QUEUE - When an already queued call reaches a converse vector step, the Delivered Event will include this reason code to inform the application that the call is still in queue. This reason applies to DELIVERED_TO_ACD only. Otherwise, this parameter will be set to AR_NONE.

originalCallInfo

[optional] Specifies the original call information. Note that information is not repeated in the originalCallInfo, if it is already reported in the CSTA service parameters or in the private data. For example, the callingDevice and calledDevice in the originalCallInfo will be NULL, if the callingDevice and the calledDevice in the CSTA service parameters are the original calling and called devices. Only when the original devices are different from the most recent callingDevice and calledDevice, the callingDevice and calledDevice in the originalCallInfo will be set. If the userEnteredCode in the private data is the original userEnteredCode, the userEnteredCode in the originalCallInfo will be UE_NONE. Only when new userEnteredCode is received and reported in the userEnteredCode, the originalCallInfo will have the original userEnteredCode.

Note: For the Delivered Event sent to the newCall of a Consultation Call, the originalCallInfo is taken from the activeCall specified in the Consultation Call request. Thus the application can pass the original call information between two calls. The calledDevice of the Consultation Call must reside on the same switch and must be monitored by the same AE Services TSAPI Service.

The original call information includes:

- **reason** - the reason for the originalCallInfo. The following reasons are supported.
  - OR_NONE - no originalCallInfo provided
  - OR_CONSULTATION - consultation call
  - OR_CONFERENCED - call conferenced
  - OR_TRANSFERRED - call transferred
  - OR_NEW_CALL - new call
- ***callingDevice*** - the original callingDevice received by the activeCall.
- ***calledDevice*** - the original calledDevice received by the activeCall.
- ***trunk*** - the original trunk group received by the activeCall. This parameter is supported by private data version 2, 3, and 4.

originalCallInfo (continued)

- ***trunkGroup*** - the original trunkGroup received by the activeCall. This parameter is supported by private data version 5 and later only.
- ***trunkMember*** (G3V4 switches and later) - the original trunkMember received by the activeCall.
- ***lookaheadInfo*** - the original lookaheadInfo received by the activeCall.
- ***userEnteredCode*** - the original userEnteredCode received by the activeCall.
- ***userInfo*** - the original userInfo received by the activeCall.

  Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo is increased to 96 bytes.

  Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.
- ***ucid*** - the original ucid of the call. This parameter is supported by private data version 5 and later only.
- ***callOriginatorInfo*** - the original callOriginatorInfo received by the activeCall. This parameter is supported by private data version 5 and later only.
- ***flexibleBilling*** - the original flexibleBilling information of the call. This parameter is supported by private data version 5 and later only.
- ***deviceHistory*** - specifies a list of deviceIDs that were previously associated with the call. For an explanation of this parameter and the following list of entries, see deviceHistory on page 544
  - **oldDeviceID (M) DeviceID**
  - **EventCause (O) EventCause**
  - **OldConnectionID (O) ConnectionID**

distributingDevice

[optional] Specifies the ACD or VDN device that distributed the call to the agent station. This information is provided only when the call is processed by the switch ACD or Call Vectoring processing and is sent for a station monitor only (that is, the delivery type is DELIVERED_TO_STATION). This parameter is supported by private data version 4 and later

The calledDevice specifies the originally called device. In most ACD call scenarios, calledDevice and distributingDevice have the same device ID. However, in call scenarios that involve call vectoring with the VDN Override feature turned on, calledDevice and distributingDevice may have different device IDs. Incoming calls arriving at the same calledDevice may be distributed to an agent via different call paths that have more than one VDN involved. If the VDN Override feature is used on the calledDevice, the distributingDevice specifies the VDN that distributes the call to the agent. This is particularly useful for applications that need to know the call path.

For example, VDN 25201 has VDN Override feature on. VDN 25201 can either route the call to VDN 25202 or VDN 25204. VDN Override is not administered on 25202 and 25204, based on conditions set up at the vector associated with VDN 25201. Both VDN 25202 and 25204 route the call to VDN 25203. Then VDN 25203 routes the call to an agent. If VDN 25201 and the agent's station are both monitored, but not VDN 25202 and 25204, the agent's station monitoring can tell from the distributingDevice whether the path of a call involves 24202 or 24204 when 25201 is called. Also note that, in the Delivered and Established events for the agent's station monitoring, the calledDevice will be 25201 and the lastRedirectionDevice will also be 25201(if VDN 25203 is monitored, the lastRedirectionDevice will change to 25203).

Proper switch administration of the VDN Override feature is required on the Communication Manager in order to receive a useful distributingDevice. The distributingDevice contains the originally called device if such administration is not performed on Communication Manager.

distributingVDN

The VDN extension associated with the distributing device. The field gets set only and exactly under the following conditions.

- When the application monitors the VDN in question and sees the C_OFFERED (translated potentially into a Delivered event, if the application does not filter it out)
- When the application monitors an agent and receives a call that came from that monitored VDN (that is, in the Delivered, Established, Transferred, and Conferenced events).

ucid

[optional] Specifies the Universal Call ID (UCID) of the resulting newCall. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

callOriginatorInfo

[optional] Specifies the callOriginatorType of the call originator such as coin call, 800-service call, or cellular call. See Table 18.

Note: CallOriginatorType values (II digit assignments) are from the network, not from Communication Manager. The II-digit assignments are maintained by the North American Numbering Plan Administration (NANPA). To obtain the most current II digit assignments and descriptions, go to:

http://www.nanpa.com/number_resource_info/ani_ii_assignments.html

flexibleBilling

[optional] Specifies whether the Flexible Billing feature is allowed for this call and the Flexible Billing customer option is assigned on the switch. If this parameter is set to TRUE, the billing rate can be changed for the incoming 900-type call using the Set Bill Rate Service. This parameter is supported by private data version 5 and later only.

deviceHistory

The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the deviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

deviceHistory (continued)

The deviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the deviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the deviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if a ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Detailed Information:

In addition to the information in the following list, see the Event Report Detailed Information section in this chapter.

- Distributing Device - There was no support for the distributingDevice parameter before Release 2.2 of Avaya Computer Telephony. In Release 2.1, the calledDevice always contains the originally called device and the distributing device, if it is different from the calledDevice, is not reported.

- In Release 1, the calledDevice contains the originally called device if there is no distributing device or contains the distributing device if call vectoring with VDN override feature of the PBX is turned on. In the later case, the originally called device is not reported.
- Last Redirection Device

Note:
There is only limited support for this parameter. An application must understand the limitations of this parameter in order to use the information correctly.

  - The accuracy of the information provided in this parameter depends on how an application monitors the devices involved in a call scenario. Experimentation may be required before an application can use this information.
  - This parameter provides the last device known by the TSAPI Service through monitor services that redirect the call or divert the call to the device (alertingDevice, answeringDevice, queued) to which the call arrives. The redirection device can be a VDN, ACD Split, or station device. The following call scenarios describe this parameter and its limitations.

Call Scenario 1:

- Both caller and agent device are monitored.
- Caller dials an ACD Split (not monitored) or a VDN (not monitored) to connect to the agent.
- Call arrives at the agent station.
  - If the caller dials the ACD Split directly, the Delivered/Established Events sent to both caller and the agent will have the ACD Split as the lastRedirectionDevice.

Note:
If the caller calls the VDN, instead of the ACD Split, and the VDN sends the call to the ACD Split, the Delivered/Established Events sent to both the caller and the agent will have the VDN as the lastRedirectionDevice. The last redirection device in the PBX is actually the ACD Split.

Note:
If the caller dials the VDN, the VDN sends the call to the ACD Split, and the call is queued at the ACD Split before the agent receives the call, the Delivered/ Established Events will have the VDN as the lastRedirectionDevice. The last redirection device in the PBX is actually the ACD Split.

Note:
If the caller calls from an external device, the agent station receives the same lastRedirectionDevice information.

Call Scenario 2:

- Both caller and agent device are monitored.
- Caller dials an ACD Split (not monitored) or a VDN (monitored) to connect to the agent.

- Call arrives at the agent station.

  Same results as in the call scenario 1, except in the following case.

  – If the caller dials the VDN, the VDN sends the call to the ACD Split, and the call is queued at the ACD Split before the agent receives the call, the Queued Event will have the VDN as the lastRedirectionDevice. The Delivered/Established Events will have the ACD Split as the lastRedirectionDevice.

  – If the caller calls from an external device, the agent station receives the same lastRedirectionDevice information.

Call Scenario 3:

- Both caller and the answering party are monitored.
- Caller dials a number (having no effect on the result whether it is monitored or not) and call goes to the first coverage point (not monitored).
- Call goes to the second coverage point (answering station).
- Call arrives at the answering station.

  – The Delivered Event sent to the caller will have the dialed number as the lastRedirectionDevice when call arrives at the first coverage point.

  – The Delivered/Established Events sent to both caller and the answering party will have the first coverage point as the lastRedirectionDevices when call arrives at the answering party.

Call Scenario 4:

- Caller is not monitored, but answering party is monitored.
- Caller dials a number (having no effect on the result whether it is monitored or not) and call goes to the first coverage point (not monitored).
- Call goes to the second coverage point (answering station).
- Call arrives at the answering station.

Note:

The Delivered/Established Events sent to the answering party will have the dialed number as the lastRedirectionDevice event though the first coverage point redirects the call to the answering party.

Call Scenario 5:

- Caller is not monitored, but answering party is monitored.
- Caller dials a number (having no effect on the result whether it is monitored or not) and call goes to the first coverage point (monitored).
- Call goes to the second coverage point (answering station).
- Call arrives at the answering station.

- The Delivered Event sent to the first coverage point will have the dialed number as the lastRedirectionDevice.
- The Delivered/Established Events sent to the answering party will have the first coverage point as the lastRedirectionDevice.
- The trunkGroup (private data version 5) trunk (private data versions 2-4), split, lookaheadInfo, userEnteredCode, and userInfo private parameters contain the most recent information about a call, while the originalCallInfo contains the original values for this information. If the most recent values are the same as the original values, the original values are not repeated in the originalCallInfo.

How to Collect userEnteredCode (UEC)

The following are steps for setting up VDNs, simple vector steps and CSTA Monitor Service requests required for a client application to receive UECs from the switch.

1. Administer a VDN and a vector on Communication Manager with a collect digits step and route command to a second VDN. See Call Scenario 1: and Call Scenario 2:.

   The purpose of this VDN is to collect UEC, but it will not report the UEC to the TSAPI Service, even if the VDN is monitored. The route command must redirect the call to a second VDN. The first VDN doesn't have to be monitored by any client application.

2. Administer a second VDN and vector to receive the redirected call from the first VDN.

   The purpose of this second VDN is to report the UEC to the TSAPI Service. Thus it must be monitored by a cstaMonitorCallsViaDevice service request from at least one client. This VDN should redirect the call to its destination. The destination can be a station extension, an ACD split, or another VDN.

   If the destination is a station extension and if the station is monitored by a cstaMonitorDevice service request, the station monitor will receive the UEC collected by the first VDN.

   If the destination is an ACD split and if an agent station in the split is monitored by a cstaMonitorDevice service request, the station monitor will receive the UEC collected by the first VDN.

   If the destination is a VDN and if the VDN is monitored by a cstaMonitorCallsViaDevice Service request, the VDN monitor will not receive the UEC collected by the first VDN.

   UEC is reported in Delivered Event Reports (for detailed information, see Call Scenario 1: and Call Scenario 2:). If multiple UECs are collected by multiple VDNs in call processing, only the most recently collected UEC is reported.

Limitations

- A monitored VDN only reports the UEC it receives (UEC collected in a previous VDN). It will not report UEC it collects or UEC collected after the call is redirected from the VDN.
- A station monitor reports only the UEC that is received by the VDN that redirects the call to the station, provided that the VDN is monitored (see Call Scenario 2:).

Call Scenario 1:

If VDN 24101 is mapped to vector 1, vector 1 has the following steps:

1. Collect 16 digits after announcement extension 1000
2. Route to 24102
3. Stop

If VDN 24102 is mapped to vector 2, vector 2 has the following steps:

1. Route to 24103

2. Stop

If 24103 is a station extension, the following can occur:

- When a call is arrived on VDN 24101, the caller will hear the announcement and the switch will wait for the caller to enter 16 digits. After the 16 digits are collected in time (if the collect digits step is timed out, the next step is executed), the call is routed to VDN 24102. The VDN 24102 routes the call to station 24103.
- If VDN 24101 is monitored using cstaMonitorCallsViaDevice, the User Entered Digits will NOT be reported in the Delivered Event Report (Call Delivered to an ACD Device) for the VDN 24101 monitor. This is because the Delivered Event Report is sent before the digits are collected.
- If VDN 24102 is monitored using cstaMonitorCallsViaDevice, the 16 digits collected by VDN 24101 will be reported in the Delivered Event Report (Call Delivered to an ACD Device) for the VDN 24102 monitor. VDN 24101 monitoring is not required for the VDN 24102 monitor to receive UEC collected by VDN 24101.
- If VDN 24102 is monitored using cstaMonitorCallsViaDevice from any client and station 24103 is monitored using cstaMonitorDevice, the 16 digits collected by VDN 24101 will be reported in the Delivered Event Report (Call Delivered to a Station Device) sent to the station 24103 monitor. If the client application is interested in the events reported by the station 24103 monitor only, call filters can be used in the cstaMonitorCallsViaDevice service to filter out all event reports from VDN 24102. This will not affect the UEC sent to the station 24103 monitor.

VDN 24102 monitoring (with or without call filters) is required for the station 24103 monitor to receive UEC collected by VDN 24101.

Call Scenario 2:

If VDN 24201 is mapped to vector 11, vector 11 has the following steps:

1. Collect 10 digits after announcement extension 2000.
2. Route to 24202.
3. Stop.

If VDN 24202 is mapped to vector 12, vector 12 has the following steps:

4. Collect 16 digits after announcement extension 3000.
5. Route to 24203.
6. Stop.

If VDN 24203 is mapped to vector 13, vector 13 has the following steps:

1. Queue to main split 2 priority.
2. Stop.

Where split 2 is a vector-controlled ACD split that has agent extensions 24301, 24302, 24303.

- When a call arrives on VDN 24201, the caller will hear an announcement and the switch will wait for the caller to enter 10 digits. After the 10 digits are collected in time, the call is routed to VDN 24202. When the call arrives on VDN 24202, the caller will hear an announcement and the switch will wait for the caller to enter 16 digits. After the 16 digits are collected in time, the call is routed to VDN 24203. The VDN 24203 queues the call to ACD Split 2. If the agent at station 24301 is available, the call is sent to station 24301.
- If VDN 24201 is monitored using cstaMonitorCallsViaDevice, the 10 digits collected by VDN 24201 will not be reported in the Delivered Event Report (Call Delivered to an ACD Device) sent for the VDN 24201 monitor. This occurs because the Delivered Event Report is sent before the digits are collected.
- If VDN 24202 is monitored using cstaMonitorCallsViaDevice, the 10 digits collected by VDN 24201 will be reported in the Delivered Event Report (Call Delivered to an ACD Device) sent for the VDN 24202 monitor.
- If VDN 24203 is monitored using cstaMonitorCallsViaDevice, the 16 digits collected by VDN 24202 will be reported in the Delivered Event Report (Call Delivered to an ACD Device) sent for the VDN 24203 monitor. However, the 10 digits collected by VDN 24201 will not be reported in the Delivered Event for the VDN 24203 monitor.
- The cstaMonitorCallsViaDevice service receives only the most recent UEC.
- If VDN 24202 and VDN 24203 are both monitored using cstaMonitorCallsViaDevice from any client, and station 24301 is monitored using cstaMonitorDevice, only the 16 digits collected by VDN 24202 will be reported in the Delivered Event Report (Call Delivered to a Station Device) for the station 24301 monitor. The cstaMonitorDevice service will receive the UEC that is received by the VDN that redirects calls to the station.

Note:

In order to receive the UEC for station monitoring, the VDN that receives the UEC and redirects calls to the station must be monitored. For example, if VDN 24203 is not monitored by any client, a cstaMonitorDevice Service on station 24301 will not receive the 16 digits collected by VDN 24202.

Table 18: Call Originator Type Values (II-digits)

Code	Description
00	Plain Old Telephone Service (POTS) - non-coin service requiring no special treatment
01	Multiparty line (more than 2) - ANI cannot be provided on 4 or 8 party lines. The presence of this "01" code will cause an Operator Number Identification (ONI) function to be performed at the distant location. The ONI feature routes the call to a CAMA operator or to an Operator Services System (OSS) for determination of the calling number.
	1 of 5

Table 18: Call Originator Type Values (ll-digits) (continued)

Code	Description
02	ANI Failure - the originating switching system indicates (by the "02" code), to the receiving office that the calling station has not been identified. If the receiving switching system routes the call to a CAMA or Operator Services System, the calling number may be verbally obtained and manually recorded. If manual operator identification is not available, the receiving switching system (e.g., an interLATA carrier without operator capabilities) may reject the call.
03-05	Unassigned
06	Station Level Rating - The "06" digit pair is used when the customer has subscribed to a class of service in order to be provided with real time billing information. For example, hotel/motels, served by PBXs, receive detailed billing information, including the calling party's room number. When the originating switching system does not receive the detailed billing information, e.g., room number, this "06" code allows the call to be routed to an operator or operator services system to obtain complete billing information. The rating and/or billing information is then provided to the service subscriber. This code is used only when the directory number (DN) is not accompanied by an automatic room/account identification.
07	Special Operator Handling Required - calls generated from stations that require further operator or Operator Services System screening are accompanied by the "07" code. The code is used to route the call to an operator or Operator Services System for further screening and to determine if the station has a denied-originating class of service or special routing/billing procedures. If the call is unauthorized, the calling party will be routed to a standard intercept message.
08-09	Unassigned
10	Not assignable - conflict with 10X test code
11	Unassigned
12-19	Not assignable - conflict with international outpulsing code
20	Automatic Identified Outward Dialing (AIOD) - without AIOD, the billing number for a PBX is the same as the PBX Directory Number (DN). With the AIOD feature, the originating line number within the PBX is provided for charging purposes. If the AIOD number is available when ANI is transmitted, code "00" is sent. If not, the PBX DN is sent with ANI code "20". In either case, the AIOD number is included in the AMA record.
21-22	Unassigned
	2 of 5

Table 18: Call Originator Type Values (II-digits) (continued)

Code	Description
23	Coin or Non-Coin - on calls using database access, e.g., 800, ANI II 23 is used to indicate that the coin/non-coin status of the originating line cannot be positively distinguished for ANI purposes by the SSP. The ANI II pair 23 is substituted for the II pairs which would otherwise indicate that the non-coin status is known, i.e., 00, or when there is ANI failure. ANI II 23 may be substituted for a valid 2-digit ANI pair on 0-800 calls. In all other cases, ANI II 23 should not be substituted for a valid 2-digit ANI II pair which is forward to an SSP from an EAEO. Some of the situations in which the ANI II 23 may be sent: ● Calls from non-conforming end offices (CAMA or LAMA types) with combined coin/ non-coin trunk groups. ● 0-800 Calls ● Type 1 Cellular Calls ● Calls from PBX Trunks ● Calls from Centrex Tie Lines
24	Code 24 identifies a toll free service call that has been translated to a Plain Old Telephone Service (POTS) routable number via the toll free database that originated for any non-pay station. If the received toll free number is not converted to a POTS number, the database returns the received ANI code along with the received toll free number. Thus, Code 24 indicates that this is a toll free service call since that fact can no longer be recognized simply by examining the called address.
25	Code 25 identifies a toll free service call that has been translated to a Plain Old Telephone Service (POTS) routable number via the toll free database that originated from any pay station, including inmate telephone service. Specifically, ANI II digits 27, 29, and 70 will be replaced with Code 25 under the above stated condition.
26	Unassigned
27	Code 27 identifies a line connected to a pay station which uses network provided coin control signaling. II 27 is used to identify this type of pay station line irrespective of whether the pay station is provided by a LEC or a non-LEC. II 27 is transmitted from the originating end office on all calls made from these lines.
28	Unassigned
29	Prison/Inmate Service - the ANI II digit pair 29 is used to designate lines within a confinement/detention facility that are intended for inmate/detainee use and require outward call screening and restriction (e.g., 0+ collect only service). A confinement/detention facility may be defined as including, but not limited to, Federal, State and/or Local prisons, juvenile facilities, immigration and naturalization confinement/detention facilities, etc., which are under the administration of Federal, State, City, County, or other Governmental agencies. Prison/Inmate Service lines will be identified by the customer requesting such call screening and restriction. In those cases where private paystations are located in confinement/ detention facilities, and the same call restrictions applicable to Prison/Inmate Service required, the ANI II digit for Prison/Inmate Service will apply if the line is identified for Prison/ Inmate Service by the customer.
	3 of 5

Table 18: Call Originator Type Values (ll-digits) (continued)

Code	Description
30-32	Intercept - where the capability is provide to route intercept calls (either directly or after an announcement recycle) to an access tandem with an associated Telco Operator Services System, the following ANI codes should be used: ● 30 Intercept (blank) - for calls to unassigned directory number (DN) ● 31 Intercept (trouble) - for calls to directory numbers (DN) that have been manually placed in trouble-busy state by Telco personnel ● 32 Intercept (regular) - for calls to recently changed or disconnected numbers
33	Unassigned
34	Telco Operator Handled Call - after the Telco Operator Services System has handled a call for an IC, it may change the standard ANI digits to "34", before outpulsing the sequence to the IC, when the Telco performs all call handling functions, e.g., billing. The code tells the IC that the BOC has performed billing on the call and the IC only has to complete the call.
35-39	Unassigned
40-49	Unrestricted Use - locally determined by carrier
50-51	Unassigned
52	Outward Wide Area Telecommunications Service (OUTWATS) - this service allows customers to make calls to a certain zone(s) or band(s) on a direct dialed basis for a flat monthly charge or for a charge based on accumulated usage. OUTWATS lines can dial station-to-station calls directly to points within the selected band(s) or zone(s). The LEC performs a screening function to determine the correct charging and routing for OUTWATS calls based on the customer's class of service and the service area of the call party. When these calls are routed to the interexchange carrier via a combined WATS-POTS trunk group, it is necessary to identify the WATS calls with the ANI code "52".
53-59	Unassigned
60	TRS - ANI II digit pair 60 indicates that the associated call is a TRS call delivered to a transport carrier from a TRS Provider and that the call originated from an unrestricted line (i.e., a line for which there are no billing restrictions). Accordingly, if no request for alternate billing is made, the call will be billed to the calling line.
61	Cellular/Wireless PCS (Type 1) - The "61" digit pair is to be forwarded to the interexchange carrier by the local exchange carrier for traffic originating from a cellular/wireless PCS carrier over type 1 trunks. (Note: ANI information accompanying digit pair "61" identifies only the originating cellular/wireless PCS system, not the mobile directory placing the call.
62	Cellular/Wireless PCS (Type 2) - The "62" digit pair is to be forwarded to the interexchange carrier by the cellular/wireless PCS carrier when routing traffic over type 2 trunks through the local exchange carrier access tandem for delivery to the interexchange carrier. (Note: ANI information accompanying digit pair "62" identifies the mobile directory number placing the call but does not necessarily identify the true call point of origin.)
	4 of 5

Table 18: Call Originator Type Values (II-digits) (continued)

Code	Description
63	Cellular/Wireless PCS (Roaming) - The "63" digit pair is to be forwarded to the interexchange carrier by the cellular/wireless PCS subscriber "roaming" in another cellular/wireless PCS network, over type 2 trunks through the local exchange carrier access tandem for delivery to the interexchange carrier. (Note: Use of "63" signifies that the "called number" is used only for network routing and should not be disclosed to the cellular/wireless PCS subscriber. Also, ANI information accompanying digit pair "63" identifies the mobile directory number forwarding the call but does not necessarily identify the true forwarded-call point of origin.)
64-65	Unassigned
66	TRS - ANI II digit pair 66 indicates that the associated call is a TRS call delivered to a transport carrier from a TRS Provider, and that the call originates from a hotel/motel. The transport carrier can use this indication, along with other information (e.g., whether the call was dialed 1+ or 0+) to determine the appropriate billing arrangement (i.e., bill to room or alternate bill).
67	TRS - ANI II digit pair 67 indicates that the associated call is a TRS call delivered to a transport carrier from a TRS Provider and that the call originated from a restricted line. Accordingly, sent paid calls should not be allowed and additional screening, if available, should be performed to determine the specific restrictions and type of alternate billing permitted.
68-69	Unassigned
70	Code 70 identifies a line connected to a pay station (including both coin and coinless stations) which does not use network provided coin control signaling. II 70 is used to identify this type pay station line irrespective of whether the pay station is provided by a LEC or a non-LEC. II 70 is transmitted from the originating end office on all calls made from these lines.
71-79	Unassigned
80-89	Reserved for Future Expansion "to" 3-digit Code
90-92	Unassigned
93	Access for private virtual network types of service: the ANI code "93" indicates, to the IC, that the originating call is a private virtual network type of service call.
94	Unassigned
95	Unassigned - conflict with Test Codes 958 and 959
96-99	Unassigned
	5 of 5

Although each value in callOriginatorType has a special meaning, neither Communication Manager nor the TSAPI Service interprets these values. The values in callOriginatorType are from the network and the application should interpret the meaning of a particular value based on The North American Numbering Plan (NANP).

Syntax

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

// CSTADeliveredEvent

typedef struct
{
      ACSHandle_t     acsHandle;
      EventClass_t    eventClass;     // CSTAUNSOLICITED
      EventType_t     eventType;      // CSTA_DELIVERED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTADeliveredEvent_t  delivered;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTADeliveredEvent_t
{
      ConnectionID_t          connection;
      SubjectDeviceID_t       alertingDevice;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      RedirectionDeviceID_t   lastRedirectionDevice;
      LocalConnectionState_t  localConnectionInfo;
      CSTAEventCause_t        cause;
} CSTADeliveredEvent_t;
```

Private Data Version 7 and 8 Syntax

If private data accompanies a CSTADeliveredEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTADeliveredEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

The deviceHistory parameter is new for private data version 7.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTDeliveredEvent - CSTA Unsolicited Event Private Data

typedef struct ATTDeliveredEvent_t {
    ATTDeliveredType_t deliveredType;
    DeviceID_t      trunkGroup;
    DeviceID_t      trunkMember;
    DeviceID_t      split;
    ATTLookaheadInfo_t lookaheadInfo;
    ATTUserEnteredCode_t userEnteredCode;
    ATTUserToUserInfo_t userInfo;
    ATTReasonCode_t reason;
    ATTOriginalCallInfo_t originalCallInfo;
    CalledDeviceID_t distributingDevice;
    ATTUCID_t       ucid;
    ATTCallOriginatorInfo_t callOriginatorInfo;
    unsigned char   flexibleBilling;
    DeviceHistory_t deviceHistory;
    CalledDeviceID_t distributingVDN;
} ATTDeliveredEvent_t;
```

Private Data Version 6 Syntax

If private data accompanies a CSTADeliveredEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTADeliveredEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV6DeliveredEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_t          eventType;// ATTV6_DELIVERED
      union
      {
          ATTV6DeliveredEvent_t v6deliveredEvent;
      } u;
} ATTEvent_t;

typedef struct ATTDeliveredEvent_t
{
      ATTDeliveredType_t      deliveredType;
      DeviceID_t              trunkGroup;
      DeviceID_t              trunkMember;
      DeviceID_t              split;
      ATTLookaheadInfo_t      lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTUserToUserInfo_t     userInfo;
      ATTReasonCode_t         reason;
      ATTOriginalCallInfo_t   originalCallInfo;
      CalledDeviceID_t        distributingDevice;
      ATTUCID_t               ucid;
      ATTCallOriginatorType_t callOriginatorInfo;
      Boolean                 flexibleBilling;
} ATTDeliveredEvent_t;

typedef enum ATTDeliveredType_t
{
      DELIVERED_TO_ACD        = 1,
      DELIVERED_TO_STATION    = 2,
      DELIVERED_OTHER         = 3 // not in use
} ATTDeliveredType_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef struct ATTLookaheadInfo_t
{
       ATTInterflow_t          type;
       ATTPriority_t           priority;
       short                   hours;
       short                   minutes;
       short                   seconds;
       DeviceID_t              sourceVDN;
       ATTUnicodeDeviceID_tuSourceVDN; // sourceVDN in Unicode
} ATTLookaheadInfo_t;

typedef enum ATTInterflow_t
{
       LAI_NO_INTERFLOW   = -1,     // indicates info not present
       LAI_ALL_INTERFLOW  = 0,
       LAI_THRESHOLD_INTERFLOW= 1,
       LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
       LAI_NOT_IN_QUEUE   = 0,
       LAI_LOW            = 1,
       LAI_MEDIUM         = 2,
       LAI_HIGH           = 3,
       LAI_TOP            = 4
} ATTPriority_t;

typedef struct ATTUnicodeDeviceID_t
{
       short                   count;
       unsigned shortvalue[64];
} ATTUnicodeDeviceID_t;

typedef struct ATTUserEnteredCode_t
{
       ATTUserEnteredCodeType_t    type;
       ATTUserEnteredCodeIndicator_tindicator;
       char                        data[ATT_MAX_USER_CODE];
DeviceID_t           collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
UE_NONE              = -1,     // indicates not specified
UE_ANY               = 0,
UE_LOGIN_DIGITS      = 2,
UE_CALL_PROMPTER     = 5,
UE_DATA_BASE_PROVIDED= 17,
UE_TONE_DETECTOR     = 32
} ATTUserEnteredCodeType_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef enum ATTUserEnteredCodeIndicator_t
{
UE_COLLECT= 0,
UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTUserToUserInfo_t
{
      ATTUUIProtocolType_ttype;
      struct {
      short       length;    // 0 indicates UUI not present
      unsigned    char    value[ATT_MAX_USER_INFO];
      } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE         = -1,    // indicates not specified
      UUI_USER_SPECIFIC= 0,     // user-specific
      UUI_IA5_ASCII    = 4      // null terminated ascii
// character string
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t
{
AR_NONE                = 0,// no reason code specified
AR_ANSWER_NORMAL       = 1,// answer supervision from
                       // the network or internal answer
AR_ANSWER_TIMED        = 2,// assumed answer based on
                           // internal timer
AR_ANSWER_VOICE_ENERGY = 3,// voice energy detection by
                           // classifier
AR_ANSWER_MACHINE_DETECTED= 4,// answering machine detected
AR_SIT_REORDER         = 5,// switch equipment congestion
AR_SIT_NO_CIRCUIT      = 6,// no circuit or channel
                           // available
AR_SIT_INTERCEPT       = 7,// number changed
AR_SIT_VACANT_CODE     = 8,// unassigned number
AR_SIT_INEFFECTIVE_OTHER= 9,// invalid number
AR_SIT_UNKNOWN         = 10,// normal unspecified
AR_IN_QUEUE            = 11,// call still in queue - for
                           // Delivered Event only
AR_SERVICE_OBSERVER= 12    // service observer connected
} ATTReasonCode_t
```

Private Data Version 6 Syntax (Continued)

```
typedef struct ATTOriginalCallInfo_t
{
      ATTReasonForCallInfo_treason;
      CallingDeviceID_t   callingDevice;
      CalledDeviceID_t    calledDevice;
      DeviceID_t          trunkGroup;
      DeviceID_t          trunkMember;
      ATTLookaheadInfo_t  lookaheadInfo;
      ATTUserEnteredCode_tuserEnteredCode;
      ATTUserToUserInfo_t userInfo;
      ATTUCID_t           ucid;
      ATTCallOriginatorType_tcallOriginatorInfo;
      Boolean             flexibleBilling;
} ATTOriginalCallInfo_t;

typedef enum ATTReasonForCallInfo_t
{
      OR_NONE         = 0,// indicates Original
                          // Call Info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;

typedef char ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t
{
      Boolean     hasInfo;    // If FALSE, no
                              // callOriginatorType
      short       callOriginatorType;
} ATTCallOriginatorInfo_t;
```

Private Data Version 5 Syntax

If private data accompanies a CSTADeliveredEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTADeliveredEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV5DeliveredEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_t          eventType;// ATT_DELIVERED
      union
      {
          ATTV5DeliveredEvent_tdeliveredEvent;
      } u;
} ATTEvent_t;

typedef struct ATTV5DeliveredEvent_t
{
      ATTDeliveredType_t      deliveredType;
      DeviceID_t              trunkGroup;
      DeviceID_t              trunkMember;
      DeviceID_t              split;
      ATTLookaheadInfo_t      lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTV5UserToUserInfo_t   userInfo;
      ATTReasonCode_t         reason;
      ATTV5OriginalCallInfo_t originalCallInfo;
      CalledDeviceID_t        distributingDevice;
      ATTUCID_t               ucid;
      ATTCallOriginatorType_t callOriginatorInfo;
      Boolean                 flexibleBilling;
} ATTV5DeliveredEvent_t;

typedef enum ATTDeliveredType_t
{
      DELIVERED_TO_ACD        = 1,
      DELIVERED_TO_STATION    = 2,
      DELIVERED_OTHER         = 3 // not in use
} ATTDeliveredType_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef struct ATTLookaheadInfo_t
{
       ATTInterflow_t          type;
       ATTPriority_t           priority;
       short                   hours;
       short                   minutes;
       short                   seconds;
       DeviceID_t              sourceVDN;
       ATTUnicodeDeviceID_tuSourceVDN; // sourceVDN in Unicode
} ATTLookaheadInfo_t;

typedef enum ATTInterflow_t
{
       LAI_NO_INTERFLOW   = -1,     // indicates info not present
       LAI_ALL_INTERFLOW  = 0,
       LAI_THRESHOLD_INTERFLOW= 1,
       LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
       LAI_NOT_IN_QUEUE   = 0,
       LAI_LOW            = 1,
       LAI_MEDIUM         = 2,
       LAI_HIGH           = 3,
       LAI_TOP            = 4
} ATTPriority_t;

typedef struct ATTUnicodeDeviceID_t
{
       short                   count;
       unsigned shortvalue[64];
} ATTUnicodeDeviceID_t;

typedef struct ATTUserEnteredCode_t
{
       ATTUserEnteredCodeType_t    type;
       ATTUserEnteredCodeIndicator_tindicator;
       char                        data[ATT_MAX_USER_CODE];
DeviceID_t           collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
UE_NONE              = -1,     // indicates not specified
UE_ANY               = 0,
UE_LOGIN_DIGITS      = 2,
UE_CALL_PROMPTER     = 5,
UE_DATA_BASE_PROVIDED= 17,
UE_TONE_DETECTOR     = 32
} ATTUserEnteredCodeType_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef enum ATTUserEnteredCodeIndicator_t
{
UE_COLLECT= 0,
UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_ttype;
      struct {
      short       length;    // 0 indicates UUI not present
      unsigned    char    value[33];
      } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE         = -1,    // indicates not specified
      UUI_USER_SPECIFIC= 0,     // user-specific
      UUI_IA5_ASCII    = 4      // null terminated ascii
// character string
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t
{
AR_NONE                = 0,// no reason code specified
AR_ANSWER_NORMAL       = 1,// answer supervision from
                       // the network or internal answer
AR_ANSWER_TIMED        = 2,// assumed answer based on
                           // internal timer
AR_ANSWER_VOICE_ENERGY = 3,// voice energy detection by
                           // classifier
AR_ANSWER_MACHINE_DETECTED= 4,// answering machine detected
AR_SIT_REORDER         = 5,// switch equipment congestion
AR_SIT_NO_CIRCUIT      = 6,// no circuit or channel
                           // available
AR_SIT_INTERCEPT       = 7,// number changed
AR_SIT_VACANT_CODE     = 8,// unassigned number
AR_SIT_INEFFECTIVE_OTHER= 9,// invalid number
AR_SIT_UNKNOWN         = 10,// normal unspecified
AR_IN_QUEUE            = 11,// call still in queue - for
                           // Delivered Event only
AR_SERVICE_OBSERVER= 12    // service observer connected
} ATTReasonCode_t
```

Private Data Version 5 Syntax (Continued)

```
typedef struct ATTV5OriginalCallInfo_t
{
      ATTReasonForCallInfo_treason;
      CallingDeviceID_t   callingDevice;
      CalledDeviceID_t    calledDevice;
      DeviceID_t          trunkGroup;
      DeviceID_t          trunkMember;
      ATTLookaheadInfo_t  lookaheadInfo;
      ATTUserEnteredCode_tuserEnteredCode;
      ATTV5UserToUserInfo_tuserInfo;
      ATTUCID_t           ucid;
      ATTCallOriginatorType_tcallOriginatorInfo;
      Boolean             flexibleBilling;
} ATTV5OriginalCallInfo_t;

typedef enum ATTReasonForCallInfo_t
{
      OR_NONE         = 0,// indicates Original
                          // Call Info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;

typedef char ATTUCID_t[64];

typedef struct ATTV5CallOriginatorInfo_t
{
      Boolean     hasInfo;    // If FALSE, no
                              // callOriginatorType
      short       callOriginatorType;
} ATTV5CallOriginatorInfo_t;
```

Private Data Version 4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV4DeliveredEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_teventType;// ATTV4_DELIVERED
      union
      {
          ATTV4DeliveredEvent_tv4deliveredEvent;
      } u;
} ATTEvent_t;

typedef struct ATTV4DeliveredEvent_t
{
      ATTDeliveredType_t  deliveredType;
      DeviceID_t          trunk;
      DeviceID_t          trunkMember;
      DeviceID_t          split;
      ATTV4LookaheadInfo_tlookaheadInfo;
      ATTUserEnteredCode_tuserEnteredCode;
      ATTV5UserToUserInfo_tuserInfo;
      ATTReasonCode_t     reason;
      ATTV4OriginalCallInfo_toriginalCallInfo;
      CalledDeviceID_t    distributingDevice;
} ATTV4DeliveredEvent_t;

typedef enum ATTDeliveredType_t
{
      DELIVERED_TO_ACD        = 1,
      DELIVERED_TO_STATION    = 2,
      DELIVERED_OTHER         = 3 // not in use
} ATTDeliveredType_t;

typedef structATTV4LookaheadInfo_t{
      ATTInterflow_t      type;
      ATTPriority_t       priority;
      short               hours;
      short               minutes;
      short               seconds;
      DeviceID_t          sourceVDN;
} ATTV4LookaheadInfo_t;
```

Private Data Version 4 Syntax (Continued)

```
typedef enum ATTInterflow_t
{
      LAI_NO_INTERFLOW= -1,// indicates info not present
      LAI_ALL_INTERFLOW= 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
      LAI_NOT_IN_QUEUE= 0,
      LAI_LOW     = 1,
      LAI_MEDIUM  = 2,
      LAI_HIGH    = 3,
      LAI_TOP     = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t     type;
      ATTUserEnteredCodeIndicator_tindicator;
      char                 data[ATT_MAX_USER_CODE];
DeviceID_t         collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
UE_NONE        = -1,   // indicates not specified
UE_ANY         = 0,
UE_LOGIN_DIGITS= 2,
UE_CALL_PROMPTER= 5,
UE_DATA_BASE_PROVIDED= 17,
UE_TONE_DETECTOR= 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
UE_COLLECT= 0,
UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_ttype;
      struct {
      short        length;    // 0 indicates UUI not present
      unsigned charvalue[33];
      } data;
} ATTV5UserToUserInfo_t;
```

Private Data Version 4 Syntax (Continued)

```
typedef enum ATTUUIProtocolType_t
{
       UUI_NONE           = -1,     // indicates not specified
       UUI_USER_SPECIFIC  = 0,      // user-specific
       UUI_IA5_ASCII      = 4       // null terminated ascii
// character string
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t
{
AR_NONE              = 0, // no reason code specified
AR_ANSWER_NORMAL     = 1, // answer supervision from
                          // the network or internal answer
AR_ANSWER_TIMED      = 2, // assumed answer based on
                          // internal timer
AR_ANSWER_VOICE_ENERGY= 3,// voice energy detection by
                          // classifier
AR_ANSWER_MACHINE_DETECTED= 4,// answering machine detected
AR_SIT_REORDER       = 5,      // switch equipment
                               // congestion
AR_SIT_NO_CIRCUIT    = 6,      // no circuit or channel
                               // available
AR_SIT_INTERCEPT     = 7,      // number changed
AR_SIT_VACANT_CODE   = 8,      // unassigned number
AR_SIT_INEFFECTIVE_OTHER = 9,  // invalid number
AR_SIT_UNKNOWN       = 10,     // normal unspecified
AR_IN_QUEUE          = 11 // call still in queue - for
// Delivered Event only} ATTReasonCode_t

typedef struct ATTV4OriginalCallInfo_t
{
       ATTReasonForCallInfo_t  reason;
       CallingDeviceID_t       callingDevice;
       CalledDeviceID_t        calledDevice;
       DeviceID_t              trunk;
       DeviceID_t              trunkMember;
       ATTV4LookaheadInfo_t    lookaheadInfo;
       ATTUserEnteredCode_t    userEnteredCode;
       ATTV5UserToUserInfo_t   userInfo;
} ATTV4OriginalCallInfo_t;

typedef enum ATTReasonForCallInfo_t
{
       OR_NONE            = 0, // indicates Original
                               // Call Info not present
       OR_CONSULTATION    = 1,
       OR_CONFERENCED     = 2,
       OR_TRANSFERRED     = 3,
       OR_NEW_CALL        = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;
```

Private Data Versions 2 and 3 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV3DeliveredEvent - CSTA Unsolicited Event Private Data

typedef struct
{
       ATTEventType_teventType;// ATTV3_DELIVERED
       union
       {
            ATTV3DeliveredEvent_tv3deliveredEvent;
       } u;
} ATTEvent_t;

typedef struct ATTV3DeliveredEvent_t
{
       ATTDeliveredType_t    deliveredType;
       DeviceID_t            trunk;
       DeviceID_t            trunkMember;
       DeviceID_t            split;
       ATTV4LookaheadInfo_t  lookaheadInfo;
       ATTUserEnteredCode_t  userEnteredCode;
       ATTV5UserToUserInfo_t userInfo;
       ATTReasonCode_t       reason;
       ATTV4OriginalCallInfo_toriginalCallInfo;
} ATTV3DeliveredEvent_t;

typedef enum ATTDeliveredType_t
{
       DELIVERED_TO_ACD      = 1,
       DELIVERED_TO_STATION  = 2,
       DELIVERED_OTHER       = 3  // not in use
} ATTDeliveredType_t;

typedef structATTV4LookaheadInfo_t
{
       ATTInterflow_t    type;
       ATTPriority_t     priority;
       short             hours;
       short             minutes;
       short             seconds;
       DeviceID_t        sourceVDN;
} ATTV4LookaheadInfo_t;

typedef enum ATTInterflow_t
{
       LAI_NO_INTERFLOW= -1,// indicates info not present
       LAI_ALL_INTERFLOW= 0,
       LAI_THRESHOLD_INTERFLOW= 1,
       LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;
```

Private Data Versions 2 and 3 Syntax (Continued)

```
typedef enum ATTPriority_t
{
      LAI_NOT_IN_QUEUE= 0,
      LAI_LOW      = 1,
      LAI_MEDIUM   = 2,
      LAI_HIGH     = 3,
      LAI_TOP      = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t      type;
      ATTUserEnteredCodeIndicator_tindicator;
      char                  data[ATT_MAX_USER_CODE];
DeviceID_t         collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
UE_NONE        = -1,    // indicates not specified
UE_ANY         = 0,
UE_LOGIN_DIGITS= 2,
UE_CALL_PROMPTER= 5,
UE_DATA_BASE_PROVIDED= 17,
UE_TONE_DETECTOR= 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
UE_COLLECT= 0,
UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_ttype;
      struct {
      short        length;    // 0 indicates UUI not present
      unsigned charvalue[33];
      } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE         = -1,    // indicates not specified
      UUI_USER_SPECIFIC= 0,     // user-specific
      UUI_IA5_ASCII    = 4      // null terminated ascii
                                // character string
} ATTUUIProtocolType_t;
```

Private Data Versions 2 and 3 Syntax (Continued)

```
typedef enum ATTReasonCode_t
{
AR_NONE                = 0,// no reason code specified
AR_ANSWER_NORMAL       = 1,// answer supervision from
                           // the network or internal answer
AR_ANSWER_TIMED        = 2,// assumed answer based on
                           // internal timer
AR_ANSWER_VOICE_ENERGY= 3,  // voice energy detection by
                           // classifier
AR_ANSWER_MACHINE_DETECTED= 4,// answering machine detected
AR_SIT_REORDER          = 5,    // switch equipment
                                // congestion
AR_SIT_NO_CIRCUIT       = 6,    // no circuit or channel
                                // available
AR_SIT_INTERCEPT        = 7,    // number changed
AR_SIT_VACANT_CODE      = 8,    // unassigned number
AR_SIT_INEFFECTIVE_OTHER = 9,   // invalid number
AR_SIT_UNKNOWN          = 10,   // normal unspecified
} ATTReasonCode_t

typedef struct ATTV4OriginalCallInfo_t
{
      ATTReasonForCallInfo_treason;
      CallingDeviceID_t    callingDevice;// original info
      CalledDeviceID_t     calledDevice;// original info
      DeviceID_t           trunk;  // original info
      DeviceID_t           trunkMember;// not in use
      ATTV4LookaheadInfo_t lookaheadInfo; // original info
      ATTUserEnteredCode_t userEnteredCode;// original info
      ATTV5UserToUserInfo_tuserInfo;// original info
} ATTV4OriginalCallInfo_t;

typedef enum ATTReasonForCallInfo_t
{
      OR_NONE      = 0,     // indicates Original
                            // Call Info not present
      OR_CONSULTATION  = 1,
      OR_CONFERENCED   = 2,
      OR_TRANSFERRED   = 3,
      OR_NEW_CALL      = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;
```

Diverted Event

Summary

- Direction: Switch to Client
- Event: *CSTADivertedEvent*
- Private Data Event: *ATTDivertedEvent* (private data version 7)
- Service Parameters: *monitorCrossRefID, connection, divertingDevice, newDestination, localConnectionInfo, cause*
- Private Parameter: *deviceHistory*

Functional Description:

The Diverted Event Report indicates a call that has been deflected or diverted from a monitored device, and is no longer present at the device.

The Diverted Event Report is sent to notify the client application that event reports for a call will no longer be provided. This event report is sent under the following circumstances:

- When a call enters a new VDN or ACD split that is being monitored.[1] For example, if a call leaves one monitored ACD device and enters another, a Call Diverted Event Report is sent to the monitor for the first ACD device. A Delivered Event Report must have been received by the ACD monitoring before the Diverted Event Report.
- When a call leaves a monitored station, without having been dropped or disconnected, this report is sent to the monitor for the station. A Delivered Event Report must have been received by the station monitoring before the Diverted Event Report.
- When a call that had been alerting at the station leaves the station because:
  - One member of a coverage and/or answer group answers a call offered to a coverage group. In this case, all other members of the coverage and/or answer group that were alerting for the call receive a Diverted Event Report.
  - A call has gone to AUDIX coverage and the Coverage Response Interval (CRI) has elapsed (the principal's call is redirected).

1. Described in the Delivered Event section.

  - The principal answers the call while the coverage point is alerting and the coverage point is dropped from the call.
  - For stations that are members of a TEG group with no associated TEG button (typically analog stations).
- The monitored station is an analog phone and an alerting call is now alerting elsewhere (gone to coverage) because:
  - The pick-up feature is used to answer a call alerting an analog principal's station.
  - An analog phone call is sent to coverage due to "no answer" (the analog station's call is redirected).

This event report will not be sent if the station is never alerted or if it retains a simulated bridge appearance until the call is dropped/disconnected. Examples of situations when this event is not sent are:

- Bridging
- Call forwarding
- Calls to a TEG (multifunction set with TEG button)
- Cover-All
- Coverage/Busy
- Incoming PCOL calls (multifunction sets)
- Pick-up for multifunction set principals

This event report will never follow an Established Event Report and is always preceded by a Delivered Event Report.

Note:

This applies to streams opened with Private Data Version 5, or later, only. If an application opens a stream with Private Data Version 4 or earlier, it will not be affected by this change. When the application opens a Private Data Version 5 stream, the Diverted Event is sent for all station device monitors, ACD devices (VDNs and ACD Splits) monitors, and call monitors independent of whether the diverting device is the monitored device. A station device monitor, an ACD device monitor, or a call monitor will be reported whether a call is leaving or staying at a previously alerted device (when a call goes to a coverage point) via the presence or absence of the Diverted Event, respectively. Note that this change only affects the Diverted event reporting; there is no private data change for the Diverted Event itself.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
connection	[mandatory] Specifies the connection that was alerting.
divertingDevice	[optional - partially supported] Specifies the device from which the call was diverted.
newDestination	[optional - partially supported] Specifies the device to which the call was diverted.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for the cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - supported] Specifies the cause for this event. The following cause is supported: ● EC_REDIRECTED - The call has been redirected.

Private Parameters

deviceHistory

The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the deviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

The deviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the deviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the deviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if a ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

// CSTADivertedEvent

typedef struct
{
      ACSHandle_tacsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_teventType;   // CSTA_DIVERTED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTADivertedEvent_t  diverted;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTADivertedEvent_t
{
      ConnectionID_t          connection;
      SubjectDeviceID_t       divertingDevice;
      CalledDeviceID_t        newDestination;
      LocalConnectionState_t  localConnectionInfo;
      CSTAEventCause_t        cause;
} CSTADivertedEvent_t;

typedef ExtendedDeviceID_tSubjectDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;
```

Private Data Version 7 and 8 Syntax

The CSTA Diverted Event includes a private data event, *ATTDivertedEvent* for private data version 7. The *ATTDivertedEvent* uses the deviceHistory private data parameter.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTDivertedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTDivertedEvent_t {
    DeviceHistory_t deviceHistory;
} ATTDivertedEvent_t;
```

Do Not Disturb Event

Summary

- Direction: Switch to Client
- Event: CSTADoNotDisturbEvent
- Service Parameters: monitorCrossRefID, device, doNotDisturbOn

Functional Description

This event report indicates a change in the status of the Do Not Disturb feature for a specific device. When the Do Not Disturb feature is active at a device, all calls to that device will be automatically forwarded to the device coverage path.

The Do Not Disturb event is available beginning with Communication Manager 5.0 and AE Services 4.1. This event is only available if the TSAPI Link is administered with ASAI Link Version 5 or later. Applications should use the cstaGetAPICaps() service to determine whether this event will be provided.

Syntax

The following structure shows only the relevant portions of the unions for this message. See ACS Data Types on page 87 and CSTA Event Data Types on page 104 for a complete description of the event structure.

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

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
             CSTAMonitorCrossRefID_t monitorCrossRefID;
                union
                {
                   CSTADoNotDisturbEvent_t  doNotDisturb,
                } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct
{
    SubjectDeviceID_t  device;
    Boolean            doNotDisturbOn;
} CSTADoNotDisturbEvent_t;
```

Parameters

acsHandle	This is the handle for the ACS Stream.
eventClass	This is a tag with the value **CSTAUNSOLICITED**, which identifies this message as an CSTA unsolicited event.
eventType	This is a tag with the value **CSTA_DO_NOT_DISTURB**, which identifies this message as an **CSTADoNotDisturbEvent**.
monitorCrossRefID	This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.
device	Specifies the device for which the Do Not Disturb feature has been activated/deactivated. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required
doNotDisturbOn	Specifies whether the DO Not Disturb feature is on (1) or off (0).

Entered Digits Event (Private)

Summary

- Direction: Switch to Client
- Event: CSTAPrivateStatusEvent
- Private Data Event: ATTEnteredDigitsEvent
- Service Parameters: monitorCrossRefID
- Private Parameters: connection, digits, localConnectionInfo, cause

Functional Description:

The Entered Digits Event is sent when a DTMF tone detector attached to a call and DTMF tones are received. The tone detector is disconnected when the far end answers or "#" is detected. The digits reported include: 0-9, "*", and "#". The digit string includes the "#", if present. Up to 24 digits can be entered.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.

Private Parameters:

connection	[mandatory] Specifies the callID of the call for which this event is reported.
digits	[mandatory] Specifies the digits user entered. The digits reported include: 0-9, "*", and "#". The digit string includes the "#", if present. The digit string is null terminated.
localConnectionInfo	[optional] Specifies the local connection state as perceived by the monitored device on this call. A value of CS_NONE is always specified.
cause	[optional] Specifies the cause for this event.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

// CSTAPrivateStatusEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass;     // CSTAUNSOLICITED
      EventType_t eventType;      // CSTA_PRIVATE_STATUS
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
          union
          {
              CSTAPrivateEvent_tprivateStatus;
          } u;

          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Parameter Syntax

If private data accompanies a CSTAPrivateStatusEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAPrivateStatusEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTEnteredDigitsEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_teventType;// ATT_ENTERED_DIGITS
      union
      {
          ATTEnteredDigitsEvent_tenteredDigitsEvent;
      } u;
} ATTEvent_t;

// ATT Entered Digits Structure
typedef struct ATTEnteredDigitsEvent_t
{
      ConnectionID_t      connection;
      char                digits[ATT_MAX_ENTERED_DIGITS];
      LocalConnectionState_tlocalConnectionInfo;
      CSTAEventCause_t    cause;
} ATTEnteredDigitsEvent_t;
```

Established Event

Summary

- Direction: Switch to Client
- Event: *CSTAEstablishedEvent*
- Private Data Event: *ATTEstablishedEvent* (private data version 7), *ATTV6EstablishedEvent* (private data version 6), *ATTV5EstablishedEvent* (private data version 5), *ATTV4EstablishedEvent* (private data version 4), *ATTV3EstablishedEvent* (private data versions 2 and 3)
- Service Parameters: *monitorCrossRefID, establishedConnection, answeringDevice, callingDevice, calledDevice, lastRedirectionDevice, localConnectionInfo, cause*
- Private Parameters: t*runkGroup, trunkMember, split, lookaheadInfo, userEnteredCode, userInfo, reason, originalCallInfo, distributingDevice, distributing VDN, ucid, callOriginatorInfo, flexibleBilling, deviceHistory*

Functional Description:

The Established Event Report indicates that the switch detects that a device has answer or connected to a call.

Before **After**

The Established Event Report is sent as follows:

- When a cstaMakePredictiveCall call is delivered to an on-PBX party (after having been answered at the destination) and the on-PBX party answers the call (picked up handset or cut-through after zip tone).
- When a cstaMakePredictiveCall call is placed to an off-PBX destination and an ISDN CONNect message is received from an ISDN-PRI facility.
- When a cstaMakePredictiveCall call is placed to an off-PBX destination and the call classifier detects an answer or a Special Information Tone (SIT) administered to answer.
- When a call is delivered to an on-PBX party and the on-PBX party has answered the call (picked up handset or cut-through after zip tone).
- When a call is redirected to an off-PBX destination, and the ISDN CONN (ISDN connect) message is received from an ISDN-PRI facility.

- Any time a station is connected to a call (picked up on a bridged call appearance, service observing, busy verification, etc.).

In general, the Established Event Report is not sent for split or vector announcements nor it is sent for the attendant group (0).

Multiple Established Event Reports

Multiple Established Event Reports may be sent for a specific call. For example, when a call is first picked up by coverage, the event is sent to the active monitors for the coverage party, as well as to the active monitors for all other extensions already on the call. If the call is then bridged onto by the principal, the Established Event Report is then sent to the monitors for the principal, as well as to the monitors for all other extensions active on the call.

Multiple Established Event Reports may also be sent for the same extension on a call. For example, when a call is first picked up by a member of a bridge, TEG, PCOL, an Established Event Report is generated. If that member goes on-hook and then off-hook again while another member of the particular group is connected on the call, a second Established Event Report will be sent for the same extension. This event report is not sent for split or vector announcements, nor it is sent for the attendant group (0).

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
establishedConnection	[mandatory] Specifies the endpoint that joined the call.
answeringDevice	[mandatory] Specifies the device that joined the call. For outgoing calls over PRI facilities -"connected number" from the ISDN CONN (ISDN connect) message. **Note:** For outgoing calls over non_PRI facilities, there is no Established Event Report. A Network Reached Event Report is generated instead. If the device being connected is on-PBX, then the extension of the device is specified (primary extension for TEGs, PCOLs, bridging).
callingDevice	[mandatory] Specifies the calling device. The following rules apply: For internal calls originated at an on-PBX station - the station's extension is specified. For outgoing calls over PRI facilities -"calling number" from the ISDN SETUP message or its assigned trunk identifier is specified, if the "calling number" does not exist (it is NULL). For incoming calls over PRI facilities -"calling number" from the ISDN SETUP message or its assigned trunk identifier is specified, if the "calling number" does not exist (it is NULL). For incoming calls over non-PRI facilities - the calling party number is generally not available. The assigned trunk identifier is provided instead. **Note:** The trunk identifier is a dynamic identifier, and it cannot be used to access a trunk in Communication Manager.

	The trunk group number is specified only when the calling party number is not available. For calls originated at a bridged call appearance -the principal's extension is specified.
calledDevice	[mandatory - partially supported] Specifies the originally called device. The following rules apply: For outgoing calls over PRI facilities - "called number" from the ISDN SETUP message is specified. If the "called number" does not exist (it is NULL), the deviceIDStatus is ID_NOT_KNOWN. For incoming calls over PRI facilities - "called number" from the ISDN SETUP message is specified. If the "called number" does not exist (it is NULL), the deviceIDStatus is ID_NOT_KNOWN. For incoming calls over non-PRI facilities - the principal extension is specified. It may be a group extension for TEG, hunt group, VDN. If the switch is administered to modify the DNIS digits, then the modified DNIS string is specified. For incoming calls to PCOL, the deviceID is ID_NOT_KNOWN. For incoming calls to a TEG (principal) group, the TEG group extension is specified. For incoming calls to a principal with bridges, the principal's extension is specified. If the called device is on-PBX and the call did not come over a PRI facility, the extension of the party dialed is specified.
lastRedirectionDevice	[optional - limited support] Specifies the previously redirection/alerted device in the case where the call was redirected/diverted to the answeringDevice.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - supported] Specifies the cause for this event. The following causes are supported: EC_TRANSFER - A call transfer has occurred. This cause has higher precedence than the following two. See Blind Transfer in the Detailed Information: section. EC_KEY_CONFERENCE - Indicates that the event report occurred at a bridged device. This cause has higher precedence than the following one EC_NEW_CALL - The call has not yet been transferred. EC_PARK - The call is connected due to picking up a parked call.

cause

EC_ACTIVE_MONITOR - This is the cause value if the Established Event Report resulted from a Single Step Conference request and the Single Step Conference request is for PT_ACTIVE. For details, see "Single Step Conference Call Service" in Chapter 4.

EC_SILENT_MONITOR -

This is the cause value if the Established Event Report resulted from a Single Step Conference request and the Single Step Conference request is for PT_SILENT. For details, see "Single Step Conference Call Service" in Chapter 4.

This is also the cause value if the Established Event Report resulted from a Service Observer (with either listen-only or listen-and-talk mode) joining the call. In this case, the reason parameter in private data version 5 and later will have AR_SERVICE_OBSERVER. Private data version 4 and earlier will not have this information.

An application cannot distinguish between case 1 from and case 2 using the cause value only. However, the reason parameter in private data version 5 and later indicates whether the EC_SILENT_MONITOR is from Single Step Conference or Service Observer. The EC_SILENT_MONITOR for AR_SERVICE_OBSERVER is a G3V6 feature.

EC_SINGLE_STEP_TRANSFER (private data version 8 or later) - The call was answered at the answeringDevice as the result of a Single Step Transfer Call operation. This cause value may occur in certain coverage scenarios where Simulated Bridging is enabled and the answering device is an extension administered with the Auto-Answer feature.

Private Parameters:

trunkGroup	[optional] Specifies the trunk group number from which the call originated. Beginning with G3V8, trunk group number is provided regardless of whether the callingDevice is available. Prior to G3V8, trunk group number is provided only if the callingDevice is unavailable. This parameter is supported by private data version 5 and later only.
trunk	[optional] Specifies the trunk group number from which the call originated. Trunk group number is provided only if the callingDevice is unavailable. This parameter is supported by private data versions 2, 3, and 4 only.
trunkMember	[optional - limited supported] This parameter is supported beginning with G3V4. It specifies the trunk member number from which the call originated. Beginning with G3V8, trunk member number is provided regardless of whether the callingDevice is available. Prior to G3V8, trunk member number is provided only if the callingDevice is unavailable.
split	[optional] Specifies the ACD split extension to which the call is delivered.
distributingDevice	[optional] Specifies the ACD or VDN device that distributed the call to the station. This information is provided only when the call was processed by the switch ACD or Call Vectoring processing and is only sent for a station monitor (i.e., the delivery type is DELIVERED_TO_STATION). This parameter is supported by private data version 4 and later.
distributingVDN	The VDN extension associated with the distributing device. The field gets set only and exactly under the following conditions. ● When the application monitors the VDN in question and sees the C_OFFERED (translated potentially into a Delivered event, if the application does not filter it out) ● When the application monitors an agent and receives a call that came from that monitored VDN (that is, in the Delivered, Established, Transferred, and Conferenced events).
lookaheadInfo	[optional] Specifies the lookahead interflow information received from the established call. The lookahead interflow is a G3 switch feature that routes some of the incoming calls from one switch to another so that they can be handled more efficiently and will not be lost. The lookahead interflow information is provided by the switch that overflows the call. A routing application may use the lookahead interflow information to determine the destination of the call. See the G3 Feature Description for more information about lookahead interflow. If the lookahead interflow type is set to "LAI_NO_INTERFLOW", no lookahead interflow private data is provided with this event.
userEnteredCode	[optional] Specifies the code/digits that may have been entered by the caller through the G3 call prompting feature or the collected digits feature. If the userEnteredCode code is set to "UE_NONE", no userEnteredCode private data is provided with this event.

userInfo

[optional] Contains user-to-user information. This parameter allows an application to associate caller information, up to 32 or 96 bytes, with a call. This information may be a customer number, credit card number, alphanumeric digits, or a binary string.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo is increased to 96 bytes.

Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE - There is no data provided in the data parameter.
- UUI_USER_SPECIFIC - The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII - The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

reason

[optional] Specifies the reason that caused this event. The following reasons are supported:

- AR_NONE- indicate no value specified for reason.
- AR_ANSWER_NORMAL- answer supervision from the network or internal answer.
- AR_ANSWER_TIMED - assumed answer based on internal timer.
- AR_ANSWER_VOICE_ENERGY - voice energy detection from a call classifier.
- AR_ANSWER_MACHINE_DETECTED - answering machine detected
- AR_SIT_REORDER - switch equipment congestion
- AR_SIT_NO_CIRCUIT - no circuit or channel available
- AR_SIT_INTERCEPT - number changed
- AR_SIT_VACANT_CODE - unassigned number
- AR_SIT_INEFFECTIVE_OTHER - invalid number
- AR_SIT_UNKNOWN - normal unspecified

originalCallInfo

[optional] Specifies the original call information. Note that information is not repeated in the originalCallInfo, if it is already reported in the CSTA service parameters or in the private data. For example, the callingDevice and calledDevice in the originalCallInfo will be NULL, if the callingDevice and the calledDevice in the CSTA service parameters are the original calling and called devices. Only when the original devices are different from the most recent callingDevice and calledDevice, the callingDevice and calledDevice in the originalCallInfo will be set. If the userEnteredCode in the private data is the original (first time entered) userEnteredCode, the userEnteredCode in the originalCallInfo will be UE_NONE. Only when new (second time entered) userEnteredCode is received, will originalCallInfo have the original userEnteredCode.

originalCallInfo (continued)

Note: For the Established Event sent to the newCall of a Consultation Call, the originalCallInfo is taken from the activeCall specified in the Consultation Call request. Thus the application can pass the original call information between two calls. The calledDevice of the Consultation Call must reside on the same switch and must be monitored via the same Tserver.

The originalCallInfo includes the original call information received by the activeCall in the Consultation Call request. The original call information includes:

reason - the reason for the originalCallInfo. The following reasons are supported.

- OR_NONE - no originalCallInfo provided
- OR_CONFERENCED - call conferenced
- OR_CONSULTATION - consultation call
- OR_TRANSFERRED - call transferred
- OR_NEW_CALL - new call

- ***callingDevice*** - the original callingDevice received by the activeCall.
- ***calledDevice*** - the original calledDevice received by the activeCall.
- ***trunk*** - the original trunk group received by the activeCall. This parameter is supported by private data versions 2, 3, and 4.
- ***trunkGroup*** - the original trunkGroup received by the activeCall. This parameter is supported by private data version 5 and later only.
- ***trunkMember*** (G3V4 switches and later) - the original trunkMember received by the activeCall.
- ***lookaheadInfo*** - the original lookaheadInfo received by the activeCall.
- ***userEnteredCode*** - the original userEnteredCode received by the activeCall.
- ***userInfo*** - the original userInfo received by the activeCall.

  Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo is increased to 96 bytes.

  Note: An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

- ***ucid*** - the original ucid of the call. This parameter is supported by private data version 5 and later only.
- ***callOriginatorInfo*** - the original callOriginatorInfo for the call. This parameter is supported by private data version 5 and later only.
- ***flexibleBilling*** - the original flexibleBilling information of the call. This parameter is supported by private data version 5 and later only.

originalCallInfo (continued)

- ***deviceHistory*** - specifies a list of deviceIDs that were previously associated with the call. For an explanation of this parameter and the following list of entries, see deviceHistory on page 592
  - **oldDeviceID (M) DeviceID**
  - **EventCause (O) EventCause**
  - **OldConnectionID (O) ConnectionID**

ucid

[optional] Specifies the Universal Call ID (UCID) of the call. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

callOriginatorInfo

[optional] Specifies the callOriginatorType of the call originator such as coin call, 800-service call, or cellular call. See Table 18.

Note: CallOriginatorType values (II digit assignments) are from the network, not from Communication Manager. The II-digit assignments are maintained by the North American Numbering Plan Administration (NANPA). To obtain the most current II digit assignments and descriptions, go to:

http://www.nanpa.com/number_resource_info/ani_ii_assignments.html

flexibleBilling

[optional] Specifies whether the Flexible Billing feature is allowed for this call and the Flexible Billing customer option is assigned on the switch. If this parameter is set to TRUE, the billing rate can be changed for the incoming 900-type call using the Set Bill Rate Service. This parameter is supported by private data version 5 and later only.

deviceHistory

The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the deviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

deviceHistory (continued)

The deviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the deviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the deviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if a ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

- Call Classification - For cstaMakePredictiveCall, the switch uses the Call Classification process, along with a variety of internal and external events, to determine a predictive (switch-classified call) call outcome. Whenever the called endpoint is external, a call classifier is used.
  - The classifier is inserted in the connection as soon as the digits have been outpulsed (sent out on a circuit). A call is classified as either answered (Established Event) or dropped (Call Cleared/Connection Cleared Event).
  - A Delivered Event is reported to the application, but it is not the final classification. "Non-classified energy" is always treated as an answer classification and reported to the application in an Established Event. A modem answer back tone results in a Call Cleared/Connection Cleared Event. Special Information Tone (SIT) detection is reported to the application as an Established Event or a Call Cleared/Connection Cleared Event, depending on the customer's administration preference. Answer Machine Detection (AMD) is reported as an Established Event or a Call Cleared/ Connection Cleared Event, depending on administration or call options.
- Last Redirection Device - There is only limited support for this parameter. An application must understand the limitations of this parameter in order to use the information correctly.
- Blind Transfer - Application designers using caller information to pop screens should refer to Transferring or conferencing a call together with screen pop information on page 45, which describes how to coordinate the passing of caller information across applications.
  - An EC_TRANSFER in the cause indicates that a blind transfer occurred before the call was established. A blind transfer is a call transfer operation that completes before the receiving party answers. Thus, when the receiving party answers, the caller and the receiving party are connected. The transferring party is not part of the connection. In terms of manual operations, it is as if the transferring party presses the transfer button to put the caller on hold, dials the receiving party, and immediately presses the transfer button again (while the call is ringing at the receiving party). Since the transfer occurs between the time the call rings at the receiving party (CSTA Delivered Event) and the time that the receiving party answers the call (CSTA Established Event), the callingDevice changes between these two events.

  Note:
  Communication Manager will not send a Transferred Event for the blind transfer operation to the receiving party before or after the Established event. An application must look in the CSTA Established Event for the callingDevice (ANI) information.
- Consultation Transfer - (Also known as "manual transfer" or "supervised transfer") - The transfer does not complete before the receiving party answers. Specifically, the transferring party and the receiving party are connected and can consult before the transfer occurs. The caller is not connected to this consultation conversation. In terms of manual operations, it is as if the transferring party presses the transfer button to put the

caller on hold, dials the receiving party, the receiving party answers, the transferring and receiving parties consult, and then the transferring party presses transfer again to transfer the call. Since the transfer occurs after the time that the receiving party answers the consultation call (after the CSTA Established Event), there is no EC_TRANSFER in the cause of the Established Event.

Note:

ANI screen pop applications should follow the guidelines described in Using Original Call Information to Pop a Screen on page 48. ANI screen pop in cases where the user does a consultation transfer manually from the telephone requires information that appears on a cstaMonitorDevice of the transferring party. If both the transferring party and the receiving party run applications that use the same TSAPI Service, then this requirement is met. To do an ANI screen pop in this case, an application must look in the CSTA Transfer Event for the ANI information. An ANI screen pop for a manual consultation transfer is done in this way at the time the call transfers, not when the consultation call rings or is answered.

Additional details and interactions are found in the Event Report Detailed Information section in this chapter. The notes above are special cases and do not reflect the recommended design.

The trunkGroup, trunk, split, lookaheadInfo, userEnteredCode, userInfo private parameters contain the most recent information about a call, while the originalCallInfo contains the original values for this information. If the most recent values are the same as the original values, the original values are not repeated in the originalCallInfo.

Syntax

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

// CSTAEstablishedEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_ESTABLISHED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTAEstablishedEvent_t  established;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAEstablishedEvent_t
{
      ConnectionID_t          establishedConnection;
      SubjectDeviceID_t       answeringDevice;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      RedirectionDeviceID_t   lastRedirectionDevice;
      LocalConnectionState_t  localConnectionInfo;
      CSTAEventCause_t        cause;
} CSTAEstablishedEvent_t;
```

Private Data Version 7 and 8 Syntax

If private data accompanies a CSTAEstablishedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAEstablishedEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

The deviceHistory parameter is new for private data version 7.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTEstablishedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTEstablishedEvent_t {
    DeviceID_t      trunkGroup;
    DeviceID_t      trunkMember;
    DeviceID_t      split;
    ATTLookaheadInfo_t lookaheadInfo;
    ATTUserEnteredCode_t userEnteredCode;
    ATTUserToUserInfo_t userInfo;
    ATTReasonCode_t reason;
    ATTOriginalCallInfo_t originalCallInfo;
    CalledDeviceID_t distributingDevice;
    ATTUCID_t       ucid;
    ATTCallOriginatorInfo_t callOriginatorInfo;
    unsigned char   flexibleBilling;
    DeviceHistory_t deviceHistory;
    CalledDeviceID_t distributingVDN;
} ATTEstablishedEvent_t;
```

Private Data Version 6 Syntax

If private data accompanies a CSTAEstablishedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAEstablishedEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV6EstablishedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
       ATTEventType_t eventType;// ATTV6_ESTABLISHED
       union
       {
            ATTV6EstablishedEvent_t v6establishedEvent;
       } u;
} ATTEvent_t;

typedef struct ATTEstablishedEvent_t
{
       DeviceID_t              trunkGroup;// most recent info
       DeviceID_t              trunkMember;// not in use
       DeviceID_t              split;    // for monitor device
                                         // (station) only
       ATTLookaheadInfo_t      lookaheadInfo;// most recent info
       ATTUserEnteredCode_t    userEnteredCode;// most recent info
       ATTUserToUserInfo_t     userInfo; // most recent info
       ATTReasonCode_t         reason;
       ATTOriginalCallInfo_t   originalCallInfo;// original info
       CalledDeviceID_t        distributingDevice; // most recent info
       ATTUCID_t               ucid;
       ATTCallOriginatorInfo_t callOriginatorInfo;
       Boolean                 flexibleBilling;
} ATTEstablishedEvent_t;

typedef struct ATTLookaheadInfo_t
{
       ATTInterflow_t          type;
       ATTPriority_t           priority;
       short                   hours;
       short                   minutes;
       short                   seconds;
       DeviceID_t              sourceVDN;
       ATTUnicodeDeviceID_t    uSourceVDN;// sourceVDN in Unicode
} ATTLookaheadInfo_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef struct ATTUnicodeDeviceID_t
{
      short           count;
      unsigned short  value[64];
} ATTUnicodeDeviceID_t;

typedef enum ATTInterflow_t
{
LAI_NO_INTERFLOW= -1,// indicates Info not present
      LAI_ALL_INTERFLOW       = 0,
      LAI_THRESHOLD_INTERFLOW = 1,
      LAI_VECTORING_INTERFLOW = 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
      LAI_NOT_IN_QUEUE        = 0,
      LAI_LOW                 = 1,
      LAI_MEDIUM              = 2,
      LAI_HIGH                = 3,
      LAI_TOP                 = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t    type;
      ATTUserEnteredCodeIndicator_tindicator;
      char                data[ATT_MAX_USER_CODE];
      DeviceID_t          collectVDN; // VDN that reports
                                      // this userEnteredCode
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE         = -1,// indicates not specified
      UE_ANY          = 0,
      UE_LOGIN_DIGITS = 2,
      UE_CALL_PROMPTER= 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR= 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT= 0,
      UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef struct ATTUserToUserInfo_t
{
      ATTUUIProtocolType_t type;
      struct {
          short           length; // 0 - UUI not present
          unsigned char   value[ATT_MAX_USER_INFO];
      } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE            = -1,// indicates not specified
      UUI_USER_SPECIFIC   = 0,// user-specific
      UUI_IA5_ASCII       = 4 // null terminated ascii
                              // character string
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t
{
      AR_NONE         = 0,// no reason code specified
      AR_ANSWER_NORMAL= 1,// answer supervision from
                          // the network or internal
                          // answer
      AR_ANSWER_TIMED= 2, // assumed answer based on
                          // internal timer
      AR_ANSWER_VOICE_ENERGY= 3,// voice energy detection
                                // by classifier
      AR_ANSWER_MACHINE_DETECTED = 4,// answering machine
                                     // detected
      AR_SIT_REORDER= 5,      // switch equipment
                              // congestion
      AR_SIT_NO_CIRCUIT= 6,   // no circuit or channel
                              // available
      AR_SIT_INTERCEPT= 7,    // number changed
      AR_SIT_VACANT_CODE= 8,  // unassigned number
      AR_SIT_INEFFECTIVE_OTHER= 9,// invalid number
      AR_SIT_UNKNOWN= 10,     // normal unspecified
      AR_IN_QUEUE     = 11,   // call still in queue - for
                              // Delivered Event only
      AR_SERVICE_OBSERVER= 12 // service observer
                              // connected
} ATTReasonCode_t
```

Private Data Version 6 Syntax (Continued)

```
typedef struct ATTOriginalCallInfo_t
{
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunkGroup;
      DeviceID_t              trunkMember;
      ATTLookaheadInfo_t      lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTUserToUserInfo_t     userInfo;
      ATTUCID_t               ucid;
ATTCallOriginatorInfo_t       callOriginatorInfo;
      Boolean                 flexibleBilling;
} ATTOriginalCallInfo_t;

typedef enum ATTReasonForCallInfo_t
{
      OR_NONE         = 0,// indicates info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;

typedef char ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t
{
      Boolean     hasInfo;// if FALSE, no callOriginatorType
      short       callOriginatorType;
} ATTCallOriginatorInfo_t;
```

Private Data Version 5 Syntax

If private data accompanies a CSTAEstablishedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAEstablishedEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV5EstablishedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
       ATTEventType_teventType;// ATT_ESTABLISHED
       union
       {
            ATTV5EstablishedEvent_testablishedEvent;
       } u;
} ATTEvent_t;

typedef struct ATTV5EstablishedEvent_t
{
       DeviceID_t              trunkGroup;// most recent info
       DeviceID_t              trunkMember;// not in use
       DeviceID_t              split;    // for monitor device
                                         // (station) only
       ATTLookaheadInfo_t      lookaheadInfo;// most recent info
       ATTUserEnteredCode_t    userEnteredCode;// most recent info
       ATTV5UserToUserInfo_t   userInfo; // most recent info
       ATTReasonCode_t         reason;
       ATTV5OriginalCallInfo_t originalCallInfo;// original info
       CalledDeviceID_t        distributingDevice; // most recent info
       ATTUCID_t               ucid;
       ATTCallOriginatorInfo_t callOriginatorInfo;
       Boolean                 flexibleBilling;
} ATTV5EstablishedEvent_t;

typedef struct ATTLookaheadInfo_t
{
       ATTInterflow_t          type;
       ATTPriority_t           priority;
       short                   hours;
       short                   minutes;
       short                   seconds;
       DeviceID_t              sourceVDN;
       ATTUnicodeDeviceID_t    uSourceVDN;// sourceVDN in Unicode
} ATTLookaheadInfo_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef struct ATTUnicodeDeviceID_t
{
      short           count;
      unsigned short  value[64];
} ATTUnicodeDeviceID_t;

typedef enum ATTInterflow_t
{
LAI_NO_INTERFLOW= -1,// indicates Info not present
      LAI_ALL_INTERFLOW       = 0,
      LAI_THRESHOLD_INTERFLOW = 1,
      LAI_VECTORING_INTERFLOW = 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
      LAI_NOT_IN_QUEUE        = 0,
      LAI_LOW                 = 1,
      LAI_MEDIUM              = 2,
      LAI_HIGH                = 3,
      LAI_TOP                 = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t    type;
      ATTUserEnteredCodeIndicator_tindicator;
      char                data[ATT_MAX_USER_CODE];
      DeviceID_t          collectVDN; // VDN that reports
                                      // this userEnteredCode
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE         = -1,// indicates not specified
      UE_ANY          = 0,
      UE_LOGIN_DIGITS = 2,
      UE_CALL_PROMPTER= 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR= 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT= 0,
      UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_t type;
      struct {
          short           length; // 0 - UUI not present
          unsigned char   value[33];
      } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
      UUI_NONE            = -1,// indicates not specified
      UUI_USER_SPECIFIC   = 0,// user-specific
      UUI_IA5_ASCII       = 4 // null terminated ascii
                              // character string
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t
{
      AR_NONE         = 0,// no reason code specified
      AR_ANSWER_NORMAL= 1,// answer supervision from
                          // the network or internal
                          // answer
      AR_ANSWER_TIMED= 2, // assumed answer based on
                          // internal timer
      AR_ANSWER_VOICE_ENERGY= 3,// voice energy detection
                                // by classifier
      AR_ANSWER_MACHINE_DETECTED = 4,// answering machine
                                     // detected
      AR_SIT_REORDER= 5,      // switch equipment
                              // congestion
      AR_SIT_NO_CIRCUIT= 6,   // no circuit or channel
                              // available
      AR_SIT_INTERCEPT= 7,    // number changed
      AR_SIT_VACANT_CODE= 8,  // unassigned number
      AR_SIT_INEFFECTIVE_OTHER= 9,// invalid number
      AR_SIT_UNKNOWN= 10,     // normal unspecified
      AR_IN_QUEUE     = 11,   // call still in queue - for
                              // Delivered Event only
      AR_SERVICE_OBSERVER= 12 // service observer
                              // connected
} ATTReasonCode_t
```

Private Data Version 5 Syntax (Continued)

```
typedef struct ATTV5OriginalCallInfo_t
{
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunkGroup;
      DeviceID_t              trunkMember;
      ATTLookaheadInfo_t      lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTV5UserToUserInfo_t   userInfo;
      ATTUCID_t               ucid;
ATTV5CallOriginatorInfo_t     callOriginatorInfo;
      Boolean                 flexibleBilling;
} ATTOriginalCallInfo_t;

typedef enum ATTReasonForCallInfo_t
{
      OR_NONE         = 0,// indicates info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;

typedef char ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t
{
      Boolean     hasInfo;// if FALSE, no callOriginatorType
      short       callOriginatorType;
} ATTCallOriginatorInfo_t;
```

Private Data Version 4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV4EstablishedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_teventType;// ATTV4_ESTABLISHED
      union
      {
          ATTV4EstablishedEvent_tv4establishedEvent;
      } u;
} ATTEvent_t;

typedef struct ATTV4EstablishedEvent_t
{
      DeviceID_t    trunk;        // most recent info
      DeviceID_t    trunkMember;  // not in use
      DeviceID_t    split;        // for monitor device
                                  // (station) only
      ATTV4LookaheadInfo_t   lookaheadInfo;// most recent info
      ATTUserEnteredCode_t   userEnteredCode;// most recent info
      ATTV5UserToUserInfo_t  userInfo;    // most recent info
      ATTReasonCode_t        reason;
      ATTV4OriginalCallInfo_t originalCallInfo;// original info
      CalledDeviceID_t       distributingDevice; // most recent
                                                 // info
} ATTV4EstablishedEvent_t;

typedef struct ATTV4LookaheadInfo_t
{
      ATTInterflow_t    type;
      ATTPriority_t     priority;
      short             hours;
      short             minutes;
      short             seconds;
      DeviceID_t        sourceVDN;
} ATTV4LookaheadInfo_t;
```

Private Data Version 4 Syntax (Continued)

```
typedef enum ATTInterflow_t
{
       LAI_NO_INTERFLOW      = -1,// indicates Info not present
       LAI_ALL_INTERFLOW     = 0,
       LAI_THRESHOLD_INTERFLOW= 1,
       LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
       LAI_NOT_IN_QUEUE= 0,
       LAI_LOW      = 1,
       LAI_MEDIUM   = 2,
       LAI_HIGH     = 3,
       LAI_TOP      = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
       ATTUserEnteredCodeType_t       type;
       ATTUserEnteredCodeIndicator_t  indicator;
       char                           data[ATT_MAX_USER_CODE];
       DeviceID_t                     collectVDN; // VDN that
                                 // reports this userEnteredCode
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
       UE_NONE           = -1,// indicates not specified
       UE_ANY            = 0,
       UE_LOGIN_DIGITS   = 2,
       UE_CALL_PROMPTER  = 5,
       UE_DATA_BASE_PROVIDED= 17,
       UE_TONE_DETECTOR= 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
       UE_COLLECT    = 0,
       UE_ENTERED    = 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTV5UserToUserInfo_t
{
       ATTUUIProtocolType_t  type;
       struct {
            short             length; // 0 indicates UUI not
                                      // present
            unsigned char     value[33];
       } data;
} ATTV5UserToUserInfo_t;
```

Private Data Version 4 Syntax (Continued)

```
typedef enum ATTUUIProtocolType_t
{
      UUI_NONE        = -1,// indicates not specified
      UUI_USER_SPECIFIC= 0,// user-specific
      UUI_IA5_ASCII= 4    // null terminated ascii
                          // character string
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t
{
AR_NONE           = 0,// no reason code specified
AR_ANSWER_NORMAL= 1,      // answer supervision from
                          // the network or internal
                          // answer
AR_ANSWER_TIMED= 2,       // assumed answer based on
                          // internal timer
AR_ANSWER_VOICE_ENERGY= 3,// voice energy detection
                          // by classifier
AR_ANSWER_MACHINE_DETECTED = 4,// answering machine detected
AR_SIT_REORDER= 5,        // switch equipment congestion
AR_SIT_NO_CIRCUIT= 6,     // no circuit or channel
                          // available
AR_SIT_INTERCEPT= 7,      // number changed
AR_SIT_VACANT_CODE= 8,    // unassigned number
AR_SIT_INEFFECTIVE_OTHER= 9,// invalid number
AR_SIT_UNKNOWN= 10,       // normal unspecified
} ATTReasonCode_t

typedef struct ATTV4OriginalCallInfo_t
{
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunk;
      DeviceID_t              trunkMember;
      ATTV4LookaheadInfo_t    lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTV5UserToUserInfo_t   userInfo;
} ATTV4OriginalCallInfo_t;
```

Private Data Version 4 Syntax (Continued)

```
typedef enum ATTReasonForCallInfo_t
{
      OR_NONE         = 0,// indicates info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;
```

Private Data Versions 2 and 3 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV3EstablishedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_teventType;// ATTV3_ESTABLISHED
      union
      {
          ATTV3EstablishedEvent_tv3establishedEvent;
      } u;
} ATTEvent_t;

typedef struct ATTV3EstablishedEvent_t
{
      DeviceID_t        trunk;       // most recent info
      DeviceID_t        trunkMember; // not in use
DeviceID_t              split;       // for monitor device
                                     // (station) only
      ATTV4LookaheadInfo_t lookaheadInfo;// most recent info
      ATTUserEnteredCode_t userEnteredCode;// most recent info
      ATTV5UserToUserInfo_t userInfo;    // most recent info
      ATTReasonCode_t       reason;
      ATTV4OriginalCallInfo_toriginalCallInfo;// original info
} ATTV3EstablishedEvent_t;

typedef struct ATTV4LookaheadInfo_t
{
      ATTInterflow_t    type;
      ATTPriority_t     priority;
      short             hours;
      short             minutes;
      short             seconds;
      DeviceID_t        sourceVDN;
} ATTV4LookaheadInfo_t;
```

Private Data Versions 2 and 3 Syntax (Continued)

```
typedef enum ATTInterflow_t
{
LAI_NO_INTERFLOW       = -1,// indicates Info not present
      LAI_ALL_INTERFLOW= 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
      LAI_NOT_IN_QUEUE= 0,
      LAI_LOW      = 1,
      LAI_MEDIUM   = 2,
      LAI_HIGH     = 3,
      LAI_TOP      = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t      type;
      ATTUserEnteredCodeIndicator_tindicator;
      char                          data[ATT_MAX_USER_CODE];
      DeviceID_t       collectVDN; // VDN that reports
                                   // this userEnteredCode
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE          = -1,// indicates not specified
      UE_ANY           = 0,
      UE_LOGIN_DIGITS  = 2,
      UE_CALL_PROMPTER = 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT= 0,
      UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_t type;
      struct {
          short        length; // 0 indicates UUI not present
          unsigned charvalue[33];
      } data;
} ATTV5UserToUserInfo_t;
```

Private Data Versions 2 and 3 Syntax (Continued)

```
typedef enum ATTUUIProtocolType_t
{
      UUI_NONE        = -1,// indicates not specified
      UUI_USER_SPECIFIC= 0,// user-specific
      UUI_IA5_ASCII= 4    // null terminated ascii
                          // character string
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t
{
      AR_NONE          = 0, // no reason code specified
      AR_ANSWER_NORMAL= 1,// answer supervision from
                          // the network or internal
                          // answer
      AR_ANSWER_TIMED= 2,   // assumed answer based on
                          // internal timer
      AR_ANSWER_VOICE_ENERGY= 3,// voice energy detection
                                // by classifier
      AR_ANSWER_MACHINE_DETECTED = 4,// answering machine
                                     // detected
      AR_SIT_REORDER    = 5,     // switch equipment congestion
      AR_SIT_NO_CIRCUIT = 6,     // no circuit or channel
                                 // available
      AR_SIT_INTERCEPT  = 7, // number changed
      AR_SIT_VACANT_CODE= 8,// unassigned number
      AR_SIT_INEFFECTIVE_OTHER= 9,// invalid number
      AR_SIT_UNKNOWN= 10,        // normal unspecified
} ATTReasonCode_t

typedef struct ATTV4OriginalCallInfo_t
{
      ATTReasonForCallInfo_t     reason;
      CallingDeviceID_t          callingDevice;
      CalledDeviceID_t           calledDevice;
      DeviceID_t                 trunk;
      DeviceID_t                 trunkMember;
      ATTV4LookaheadInfo_t       lookaheadInfo;
      ATTUserEnteredCode_t       userEnteredCode;
      ATTV5UserToUserInfo_t      userInfo;
} ATTV4OriginalCallInfo_t;
```

Private Data Versions 2 and 3 Syntax (Continued)

```
typedef enum ATTReasonForCallInfo_t
{
      OR_NONE         = 0,// indicates info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;
```

Failed Event

Summary

- Direction: Switch to Client
- Event: *CSTAFailedEvent*
- Private Data Event: *ATTFailedEvent* (private data version 8), *ATTV7FailedEvent* (private data version 7)
- Service Parameters: *monitorCrossRefID, failedConnection, failingDevice, calledDevice, localConnectionInfo, cause*
- Private Parameters: *deviceHistory, callingDevice*

Functional Description:

The Failed Event Report indicates that a call cannot be completed.

Before **After**

This event report is generated when the destination of a call is busy or unavailable, as follows:

- When a call is delivered to an on-PBX station and the station is busy (without coverage and call waiting).
- When a call tries to terminate on an ACD split without going through a vector and the destination ACD split's queue is full, and the ACD split does not have coverage.
- When a call encounters a busy vector command in vector processing.
- When a Direct-Agent call tries to terminate an on-PBX ACD agent and the specified ACD agent's split queue is full and the specified ACD agent does not have coverage.
- When a call is trying to reach an off-PBX party and an ISDN DISConnect message with a User Busy cause is received from an ISDN-PRI facility.

The Failed Event Report is also generated when the destination of a call receives reorder/denial treatment, as follows:

- When a call is trying to terminate to an on-PBX destination but the destination specified is inconsistent with the dial plan, has failed the "class of restriction" check, or inter-digit timeout has occurred.
- When a call encounters a step in vector processing which causes the denial treatment to be applied to the originator.

- When a Direct-Agent call is placed to a destination agent who is not a member of the specified split.
- When a Direct-Agent call is placed to a destination agent who is not logged in.

The Failed Event Report is not sent under the following circumstances:

- For a cstaMakePredictiveCall call when any of the above conditions occur the Call Cleared Event Report is generated to indicate that the call has been terminated.

  The call is terminated because a connection could not be established to the destination.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
failingConnection	[mandatory - partially supported] Specifies the callID of the call that failed
failingDevice	[mandatory - partially supported] Specifies the device that failed. The deviceIDStatus may be ID_NOT_KNOWN.
calledDevice	[mandatory - partially supported] Specifies the called device. The following rules apply: ● For outgoing calls over PRI facilities, the "called number" from the ISDN SETUP message is specified. If the "called number" does not exist (it is NULL), the deviceIDStatus is ID_NOT_KNOWN. ● For outgoing calls over non-PRI facilities, then the deviceIDStatus is ID_NOT_KNOWN. ● For calls to a TEG (principal) group, the TEG group extension is provided. ● If the busy party is on the PBX, then the extension of the party will be specified. If there is an internal error in the extension, then the deviceIDStatus is ID_NOT_KNOWN. ● For incoming calls to a principal with bridges, the principal's extension is provided. ● If the destination is inconsistent with the dial plan, then the deviceIDStatus is ID_NOT_KNOWN.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for the cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - supported] Specifies the reason for this event. The following Event Causes are explicitly sent from the switch: ● EC_BUSY - User is busy or queue is full. ● EC_CALL_NOT_ANSWERED - User is not responding. ● EC_TRUNKS_BUSY - No trunks are available.

cause (continued)

- EC_RESOURCES_NOT_AVAILABLE- Call cannot be completed due to switching resources limitation; for example, no circuit or channel is available.
- EC_REORDER_TONE - Call is rejected or outgoing call is barred.
- EC_DEST_NOT_OBTAINABLE - Invalid destination number.
- EC_NETWORK_NOT_OBTAINABLE - Bearer capability is not available.
- EC_INCOMPATIBLE_DESTINATION - Incompatible destination number. For example, a call from a voice station to a data extension.
- EC_NO_AVAILABLE_AGENTS - Queue full or for direct agent calls - the agent is not a member of the split or the agent is not logged in.

Private Parameters

deviceHistory

The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the deviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

callingDevice

Specifies the calling device. The following rules apply:

- For internal calls - the originator's extension.
- For outgoing calls over PRI facilities - "calling number" from the ISDN SETUP message or its assigned trunk identifier is specified. If the "calling number" does not exist, it is NULL.
- For incoming calls over PRI facilities - "calling number" from the ISDN SETUP message or its assigned trunk identifier is specified. If the "calling number" does not exist, it is NULL.
- For incoming calls over non-PRI facilities - the calling party number is generally not available. The assigned trunk identifier is provided instead.

  Note: The trunk identifier is a dynamic device identifier and it can not be used to access a trunk in Communication Manager.
- The trunk identifier is specified only when the calling party number is not available.
- For calls originated at a bridged call appearance - the principal's extension is specified.

device history (continued)

The deviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the deviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the deviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if a ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

// CSTAFailedEvent

typedef struct
```

```
{
      ACSHandle_t acsHandle;
      EventClass_t eventClass;// CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_FAILED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_t eventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_t monitorCrossRefId;
              union
              {
                  CSTAFailedEvent_t  failed;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAFailedEvent_t
{
      ConnectionID_t          failedConnection;
      SubjectDeviceID_t       failingDevice;
      CalledDeviceID_t        calledDevice;
      LocalConnectionState_t  localConnectionInfo;
      CSTAEventCause_t        cause;
} CSTAFailedEvent_t;
```

Private Data Version 8 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTFailedEvent - CSTA Unsolicited Event Private Data
typedef struct ATTFailedEvent_t {
    DeviceHistory_t deviceHistory;
    CallingDeviceID_t callingDevice;
} ATTFailedEvent_t;
```

Private Data Version 7 Syntax

The CSTA Failed Event includes a private data event, *ATTV7FailedEvent* for private data version 7. The *ATTV7FailedEvent* uses the deviceHistory private data parameter.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV7FailedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTV7FailedEvent_t {
    DeviceHistory_t deviceHistory;
} ATTV7FailedEvent_t;
```

Forwarding Event

Summary

- Direction: Switch to Client
- Event: CSTAForwardingEvent
- Service Parameters: monitorCrossRefID, device, forwardingInformation

Functional Description

This event report indicates a change in the state of the Forwarding feature for a specific device. The event also indicates the type of forwarding being invoked when the feature is activated.

The Forwarding event is available beginning with Communication Manager 5.0 and AE Services 4.1. This event is only available if the TSAPI Link is administered with ASAI Link Version 5 or later. Applications should use the cstaGetAPICaps() service to determine whether this event will be provided.

Currently, AE Services does not provide the forwarding destination in the Forwarding event. However, applications may use the cstaQueryForwarding() service to determine the forwarding destination for a device where call forwarding is active.

Syntax

The following structure shows only the relevant portions of the unions for this message.

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

typedef struct
   {
       ACSEventHeader_t eventHeader;
       union
       {
          struct
          {
            CSTAMonitorCrossRefID_tmonitorCrossRefID;
              union
              {
                CSTAForwardingEvent_t  forwarding;
          }   u;
          } cstaUnsolicited;
       } event;
} CSTAEvent_t;

typedef struct
    {
        SubjectDeviceID_t device;
        ForwardingInfo_t forwardingInformation;
    } CSTAForwardingEvent_t;

typedef enum ForwardingType_t {
        FWD_IMMEDIATE = 0,
        FWD_BUSY = 1,
        FWD_NO_ANS = 2,
        FWD_BUSY_INT = 3,
        FWD_BUSY_EXT = 4,
        FWD_NO_ANS_INT = 5,
        FWD_NO_ANS_EXT = 6
    } ForwardingType_t;

typedef struct ForwardingInfo_t {
        ForwardingType_t forwardingType;
        Boolean          forwardingOn;
        DeviceID_t       forwardDN;/* NULL for not present */
    } ForwardingInfo_t;
```

Parameters

acsHandle	This is the handle for the ACS Stream.
eventClass	This is a tag with the value CSTAUNSOLICITED, which identifies this message as an CSTA unsolicited event.
eventType	This is a tag with the value **CSTA_FORWARDING** which identifies this message as an **CSTAForwardingEvent.**
monitorCrossRefID	This parameter contains the handle to the CSTA association for which this event is associated. This handle is typically chosen by the switch and should be used by the application as a reference to a specific established association.
device	Specifies the device for which the Forwarding feature has been activated/ deactivated. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required.
forwardingType	Specifies the type of forwarding being invoked for the specific device. This may include one of the following: ● **Immediate** - Forwarding all calls ● **Busy** - Forwarding when busy ● **No Answer** - Forwarding after no answer ● **Busy Internal** - Forwarding when busy for an internal call ● **Busy External** - Forwarding when busy for an external call ● **No Answer Internal** - Forwarding after no answer for an internal call ● **No Answer External** - after no answer for an external call. **Note:** AE Services, supports only the **Immediate** forwardingType.
forwardingOn	Specifies whether the Forward feature is on (1) or off (0).
forwardDN	Specifies the destination device to which the calls are being forwarded. If the device is not specified, then the parameter will indicate that the device was not known or that it was not required. **Note:** AE Services always provides a null Device ID for this parameter

Held Event

Summary

- Direction: Switch to Client
- Event: CSTAHeldEvent
- Service Parameters: monitorCrossRefID, heldConnection, holdingDevice, localConnectionInfo, cause

Functional Description:

The Held Event Report indicates that an on-PBX station has placed a call on hold. This includes the hold for conference and transfer.

Placing a call on hold can be done either manually at the station or via a Hold Service request.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
heldConnection	[mandatory] Specifies the endpoint where hold was activated.
holdingDevice	[mandatory] Specifies the station extension that placed the call on hold.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - supported] Specifies the cause for this event. The following causes are supported. • EC_KEY_CONFERENCE - Indicates that the event report occurred at a bridged device. • EC_NEW_CALL - The call has not yet been transferred.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

// CSTAHeldEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_HELD
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTAHeldEvent_t  held;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAHeldEvent_t
{
      ConnectionID_t          heldConnection;
      SubjectDeviceID_t       holdingDevice;
      LocalConnectionState_t  localConnectionInfo;
      CSTAEventCause_t        cause;
} CSTAHeldEvent_t;
```

Logged Off Event

Summary

- Direction: Switch to Client
- Event: CSTALoggedOffEvent
- Private Data Event: ATTLoggedOffEvent
- Service Parameters: monitorCrossRefID, agentDevice, agentID, agent
- Group Private Parameters: reasonCode

Functional Description:

The Logged Off Event Report informs the application that an agent has logged out of an ACD Split. An application needs to request a cstaMonitorDevice on the ACD Split in order to receive this event.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
agentDevice	[mandatory] Indicates the extension of the agent that is logging on.
agentID	[optional - not supported] Indicates the agent identifier.
agentGroup	[optional - supported] Indicates the ACD Split that is being logged on.

Private Parameters:

reasonCode	[optional] Specifies the reason for change of work mode to WM_AUX_WORK or the logged-out (AM_LOG_OUT) state. For private data version 7 valid reason codes range from 0 to 99. A value of 0 indicates that the reason code is not available. The meaning of the codes 1 through 99 is defined by the application. This range of reason codes is supported by private data version 7 only. Private data versions 6 and 5 support single digit reason codes 1 through 9. A value of 0 indicates that the reason code is not available. The meaning of the code (1-9) is defined by the application. Private data version 4 and earlier do not support reason codes.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

 // CSTALoggedOffEvent

 typedef struct
 {
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_LOGGED_OFF
 } ACSEventHeader_t;

 typedef struct
 {
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTALoggedOffEvent_t  loggedOff;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
 } CSTAEvent_t;

 typedef struct CSTALoggedOffEvent_t
 {
      SubjectDeviceID_t   agentDevice;
      AgentID_t           agentID;
      AgentGroup_t        agentGroup;
 } CSTALoggedOffEvent_t;

 typedef ExtendedDeviceID_t SubjectDeviceID_t;

 typedef char         AgentID_t[32];

 typedef DeviceID_t   AgentGroup_t;
```

Private Parameter Syntax

If private data accompanies a CSTALoggedOffEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTALoggedOffEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

 // ATTLoggedOffEvent - CSTA Unsolicited Event Private Data

 typedef struct
 {
      ATTEventType_teventType;// ATT_LOGGED_OFF
      union
      {
          ATTLoggedOffEvent_tloggedOff;
      } u;
 } ATTEvent_t;

 typedef struct ATTLoggedOffEvent_t
 {
      long    reasonCode;// single digit 1 - 9
 } ATTLoggedOffEvent_t;
```

Logged On Event

Summary

- Direction: Switch to Client
- Event: CSTALoggedOnEvent
- Private Data Event: ATTLoggedOnEvent
- Service Parameters: monitorCrossRefID, agentDevice, agentID, agentGroup, password
- Private Parameters: workMode

Functional Description:

The Logged On Event Report informs the application that an agent has logged into an ACD Split. An application needs to request a cstaMonitorDevice on the ACD Split in order to receive this event.

The initial agent work mode is provided in the private data.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
agentDevice	[mandatory] Indicates the station extension of the agent that is logging on.
agentID	[optional - partially supported] Indicates the logical agent identifier. This is provided for an EAS environment only. For a traditional ACD environment, this is not supported.
agentGroup	[optional - supported] Indicates the ACD Split that is being logged on.
password	[optional - not supported] Indicates the agent password for logging in.

Private Parameters:

workMode	[optional - not supported] Specifies the initial work mode for the Agent as Auxiliary-Work Mode (WM_AUX_WORK), After-Call-Work Mode (WM_AFT_CALL), Auto-In Mode (WM_AUTO_IN), or Manual-In-Work Mode (WM_MANUAL_IN).

Detailed Information:

In addition to the information provided below, see the Event Report Detailed Information section in this chapter.

- Service Availability - This event is only available on Communication Manager with G3V4 or later software.

Syntax

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

 // CSTALoggedOnEvent

 typedef struct
 {
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_LOGGED_ON
 } ACSEventHeader_t;

 typedef struct
 {
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTALoggedOnEvent_t  loggedOn;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
 } CSTAEvent_t;

 typedef struct CSTALoggedOnEvent_t
 {
      SubjectDeviceID_t   agentDevice;
      AgentID_t           agentID;
      AgentGroup_t        agentGroup;
      AgentPassword_t     password;// not supported
} CSTALoggedOnEvent_t;

 typedef ExtendedDeviceID_t   SubjectDeviceID_t;

 typedef char                 AgentID_t[32];
```

```
typedef DeviceID_t          AgentGroup_t;

typedef char                AgentPassword_t[32];
```

Private Parameter Syntax

If private data accompanies a CSTALoggedOnEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTALoggedOnEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTLoggedOnEvent - CSTA Unsolicited Event Private Data

typedef struct
{
     ATTEventType_teventType;// ATT_LOGGED_ON
     union
     {
         ATTLoggedOnEvent_tloggedOnEvent;
     } u;
} ATTEvent_t;

typedef struct ATTLogedOnEvent_t
{
     ATTWorkMode_tworkMode;
} ATTLoggedOnEvent_t;

typedef enum ATTWorkMode_t {
     WM_AUX_WORK         = 1,
     WM_AFTCAL_WK        = 2,
     WM_AUTO_IN          = 3,
     WM_MANUAL_IN        = 4
} ATTWorkMode_t;
```

Network Reached Event

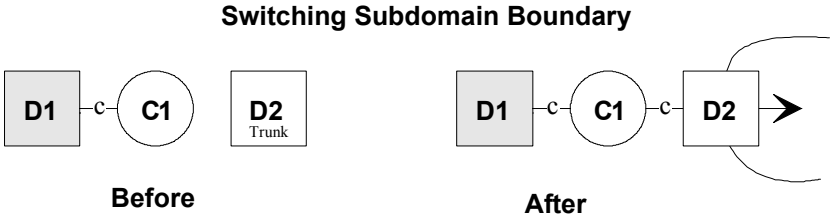
Summary

- Direction: Switch to Client
- Event: CSTANetworkReachedEvent
- Private Data Event: ATTNetworkReachedEvent (private data version 7), ATTV6NetworkReachedEvent (private data version 5 and 6), ATTV4NetworkReachedEvent (private data versions 2, 3, and 4)
- Service Parameters: monitorCrossRefID, connection, trunkUsed, calledDevice, localConnectionInfo, cause
- Private Parameters: progressLocation, progressDescription, trunkGroup, trunkMember, deviceHistory

Functional Description:

This event indicates the following two situations when establishing a connection:

- a non-ISDN call is cut through the switch boundary to another network (set to outgoing trunk), or
- an ISDN call is leaving the ISDN network.



This event report implies that there will be a reduced level of event reporting and possibly no additional device feedback, except disconnect/drop, provided for this party in the call. A Network Reached Event Report is never sent for calls made to devices connected directly to the switch.

The Network Reached Event Report is generated when:

- an ISDN PROG (ISDN progress) message has been received for a call using the ISDN-PRI facilities. The reason for the PROG (progress) message is contained in the Progress Indicator. This indicator is sent in private data.
- a call is placed to an off-PBX destination and a non-PRI trunk is seized
- a call is redirected to an off-PBX destination and a non-PRI trunk is seized.

A switch may receive multiple PROGress messages for any given call; each will generate a Network Reached Event Report. This event will not be generated for a cstaMakePredictiveCall call.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
connection	[mandatory] Specifies the endpoint for the outbound connection to another network.
trunkUsed	[mandatory - not supported] Specifies the trunk identifier that was used to establish the connection. This information is provided in the private data.
calledDevice	[mandatory - partially supported] Specifies the destination device of the call. The deviceIDStatus may be ID_NOT_KNOWN.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional -supported] Specifies the cause for this event. The following cause is supported. ● EC_REDIRECTED - The call has been redirected. ● EC_SINGLE_STEP_TRANSFER (private data version 8 or later) - The call was placed to an off-PBX destination as the result of a Single Step Transfer Call operation.

Private Parameters:

progressLocation	[mandatory] Specifies the progress location in a Progress Indicator Information Element from the PRI network. The following location indicators are supported: ● PL_USER ● PL_PUB_LOCAL ● PL_PUB_REMOTE ● PL_PRIV_REMOTE
progressDescription	[mandatory] Specifies the progress description in a Progress Indicator Information Element from the PRI network. The following description indicators are supported: ● PD_CALL_OFF_ISDN ● PD_DEST_NOT_ISDN ● PD_ORIG_NOT_ISDN ● PD_CALL_ON_ISDN

- PD_INBAND

trunkGroup [optional - limited supported] This parameter is supported by G3V6 and later switches only. Specifies the trunk group number from which the call leaves the switch and enters the network. This information will not be reported in the originalCallInfo parameter in the events following Network Reached. This parameter is supported by private data version 5 and later only.

trunkMember [optional - limited supported] This parameter is supported by G3V6 and later switches only. Specifies the trunk member from which the call leaves the switch and enters the network. This information will not be reported in the originalCallInfo parameter in the events following Network Reached. This parameter is supported by private data version 5 and later only.

deviceHistory The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the deviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

deviceHistory (continued)

The deviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the deviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the deviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if a ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

// CSTANetworkReachedEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_NETWORK_REACHED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_t monitorCrossRefId;
              union
              {
              CSTANetworkReachedEvent_t  networkReached;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTANetworkReachedEvent_t
{
      ConnectionID_t          connection;
      SubjectDeviceID_t       trunkUsed;
      CalledDeviceID_t        calledDevice;
      LocalConnectionState_t  localConnectionInfo;
      CSTAEventCause_t        cause;
} CSTANetworkReachedEvent_t;
```

Private Data Version 7 and 8 Syntax

If private data accompanies a CSTANetworkReachedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTANetworkReachedEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTNetworkReachedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTNetworkReachedEvent_t {
    ATTProgressLocation_t progressLocation;
    ATTProgressDescription_t progressDescription;
    DeviceID_t      trunkGroup;
    DeviceID_t      trunkMember;
    DeviceHistory_t deviceHistory;
} ATTNetworkReachedEvent_t;
```

Private Data Version 5 and 6 Syntax

If private data accompanies a CSTANetworkReachedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTANetworkReachedEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV6NetworkReachedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_teventType;// ATTV6_NETWORK_REACHED
      union
      {
          ATTV6NetworkReachedEvent_t v6networkReachedEvent;
      } u;
} ATTEvent_t;

typedef struct ATTNetworkReachedEvent_t
{
      ATTProgressLocation_t   progressLocation;
      ATTProgressDescription_tprogressDescription;
      DeviceID_t              trunkGroup;
      DeviceID_t              trunkMember;
} ATTNetworkReachedEvent_t;

// ATT progress location values

typedef enum ATTProgressLocation_t
{
      PL_USER         = 0,// user
      PL_PUB_LOCAL    = 1,// public network serving
                         // local user
      PL_PUB_REMOTE   = 4,// public network serving
                         // remote user
      PL_PRIV_REMOTE  = 5 // private network serving
                         // remote user
} ATTProgressLocation_t;
```

Private Data Version 5 Syntax (Continued)

```
// ATT progress description values

typedef enum ATTProgressDescription_t
{
      PD_CALL_OFF_ISDN= 1,// call is not end-to-end ISDN,
                          // call progress in-band
      PD_DEST_NOT_ISDN= 2,// destination address is
                          // non-ISDN
      PD_ORIG_NOT_ISDN= 3,// origination address is non-ISDN
      PD_CALL_ON_ISDN = 4,// call has returned to ISDN
      PD_INBAND       = 8 // in-band information now
                          // available
} ATTProgressDescription_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV4NetworkReachedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
      ATTEventType_teventType;// ATTV4_NETWORK_REACHED
      union
      {
      ATTV4NetworkReachedEvent_tv4networkReachedEvent;
      } u;
} ATTEvent_t;

typedef struct ATTV4NetworkReachedEvent_t
{
      ATTProgressLocation_t      progressLocation;
      ATTProgressDescription_t   progressDescription;
} ATTV4NetworkReachedEvent_t;

// ATT progress location values

typedef enum ATTProgressLocation_t
{
      PL_USER           = 0,     // user
      PL_PUB_LOCAL      = 1,     // public network serving
                                 // local user
      PL_PUB_REMOTE     = 4,     // public network serving
                                 // remote user
      PL_PRIV_REMOTE    = 5      // private network serving
                                 // remote user
} ATTProgressLocation_t;

// ATT progress description values

typedef enum ATTProgressDescription_t
{
      PD_CALL_OFF_ISDN= 1,   // call is not end-to-end ISDN,
                             // call progress in-band
      PD_DEST_NOT_ISDN= 2,   // destination address is
                             // non-ISDN
      PD_ORIG_NOT_ISDN= 3,   // origination address is non-ISDN
      PD_CALL_ON_ISDN   = 4, // call has returned to ISDN
      PD_INBAND         = 8  // in-band information now
                             // available
} ATTProgressDescription_t;
```

Originated Event

Summary

- Direction: Switch to Client
- Event: CSTAOriginatedEvent
- Private Data Event: ATTOriginatedEvent (private data version 6), ATTV5OriginatedEvent (private data version 2, 3, 4, and 5)
- Service Parameters: monitorCrossRefID, originatedConnection, callingDevice, calledDevice, localConnectionInfo, cause
- Private Parameters: logicalAgent, userInfo

Functional Description:

The Originated Event Report indicates that a station has completed dialing and the switch has decided to attempt the call. This event is reported to cstaMonitorDevice associations only.

Before **After**

This event is generated as follows:

- When a station user completes dialing a valid number.
- When a cstaMakeCall is requested on a station, and the station is in the off-hook state (goes off-hook manually, or is forced off-hook), the switch processes the request and determines that a call is to be attempted.
- When a call is attempted using an outgoing trunk and the switch stops collecting digits for that call.

This event will not be reported when a call is aborted because an invalid number was provided, or because the originating number provided is not allowed (via COR) to originate a call.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
originatedConnection	[mandatory] Specifies the connection for which the call has been originated.
callingDevice	[mandatory] Specifies the device from which the call has been originated.
calledDevice	[mandatory] Specifies the number that the user dialed or the destination requested by a cstaMakeCall. This is the number dialed rather than the number out-pulsed. It does not include the AAR/ARS FAC (Feature Access Code), or TAC (Trunk Access Code; for example, without the leading 9 often used as the ARS FAC).
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This information is provided for cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - supported] Specifies the cause for this event. The following causes are supported: • EC_KEY_CONFERENCE - Indicates that the event report occurred at a bridged device. This cause has higher precedence than the following cause. • EC_NEW_CALL - The call has not yet been redirected.

Private Parameters:

logicalAgent	[optional] Specifies the logical agent extension of the agent that is logged into the station making the call for a cstaMakeCall request.
userInfo	[optional] This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. This information may be a customer number, credit card number, alphanumeric digits, or a binary string. The userInfo parameter is defined for this event, but it is not supported by the DEFINITY switch (i.e., the userInfo parameter will not be received for this event).

Detailed Information:

In addition to the information provided below, see the Event Report Detailed Information section in this chapter.

- Abbreviated Dialing - The Originated Event will be reported when a call is attempted after requesting an abbreviated or speed dialing feature.
- Account Codes - (CDR or SMDR Account Code Dialing) - The Originated Event will be reported when a call is originated after an optional or mandatory account code entry.
- Authorization Codes - The Originated Event will be reported when a call is originated after an authorization code entry.
- Automatic Callback - The Originated Event will be reported when an automatic callback feature matures and the caller goes off-hook on the automatic callback call.
- Bridged Call Appearance - The Originated Event will be reported for a call originated from a bridged appearance.
- Call Park - The Originated Event will not be reported when a call is parked or retrieved from a parking spot.
- cstaMakePredictiveCall - The Originated Event will not be reported for a cstaMakePredictiveCall.
- Service Availability - This event is only available on Communication Manager with G3V4 or later software.

Syntax

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

// CSTAOriginatedEvent

typedef struct
{
      ACSHandle_t     acsHandle;
      EventClass_t    eventClass; // CSTAUNSOLICITED
      EventType_t     eventType;  // CSTA_ORIGINATED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTAOriginatedEvent_t  originated;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTAOriginatedEvent_t
{
      ConnectionID_t          originatedConnection;
      SubjectDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      LocalConnectionState_t  localConnectionInfo;
      CSTAEventCause_t        cause;
} CSTAOriginatedEvent_t;
```

Private Data Version 6 Syntax

If private data accompanies a CSTAOriginatedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAOriginatedEvent does not deliver privatedata to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTOriginatedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
       ATTEventType_t    eventType;    // ATT_ORIGINATED
       union
       {
            ATTOriginatedEvent_t       originatedEvent;
       } u;
} ATTEvent_t;

typedef struct ATTOriginatedEvent_t
{
       DeviceID_t             logicalAgent;
       ATTUserToUserInfo_t    userInfo;
} ATTOriginatedEvent_t;

typedef struct  ATTUserToUserInfo_t  {
       ATTUUIProtocolType_t  type;
       struct {
            short             length;  // 0 indicates UUI not
                                       // present
         unsigned char        value[33];
       } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
       UUI_NONE          = -1,// indicates not specified
       UUI_USER_SPECIFIC = 0, // user-specific
       UUI_IA5_ASCII     = 4  // null terminated ascii
                              // character string
} ATTUUIProtocolType_t;
```

Private Data Version 2-5 Syntax

If private data accompanies a CSTAOriginatedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAOriginatedEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTOriginatedEvent - CSTA Unsolicited Event Private Data

typedef struct
{
       ATTEventType_t    eventType;    // ATT_ORIGINATED
       union
       {
            ATTOriginatedEvent_t       originatedEvent;
       } u;
} ATTEvent_t;

typedef struct ATTOriginatedEvent_t
{
       DeviceID_t             logicalAgent;
       ATTUserToUserInfo_t    userInfo;
} ATTOriginatedEvent_t;

typedef struct  ATTV5UserToUserInfo_t  {
       ATTUUIProtocolType_t  type;
       struct {
            short             length;  // 0 indicates UUI not
                                       // present
        unsigned char         value[33];
       } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
       UUI_NONE          = -1,// indicates not specified
       UUI_USER_SPECIFIC = 0, // user-specific
       UUI_IA5_ASCII     = 4  // null terminated ascii
                              // character string
} ATTUUIProtocolType_t;
```

Queued Event

Summary

- Direction: Switch to Client
- Event: CSTAQueuedEvent
- Private Data Event: *ATTQueuedEvent* (private data version 7)
- Service Parameters: monitorCrossRefID, queuedConnection, queue, callingDevice, calledDevice, lastRedirectionDevice, numberQueued, localConnectionInfo, cause
- Private Parameters: *deviceHistory*

Functional Description:

The Queued Event Report indicates that a call queued.

Before **After**

The Queued Event report is generated as follows:

- When a cstaMakePredictiveCall call is delivered to a hunt group or ACD split and the call queues.
- When a call is delivered or redirected to a hunt group or ACD split and the call queues.

It is possible to have multiple Queued Event Reports for a call. For example, the call vectoring feature may queue a call in up to three ACD splits at any one time. In addition, the event is sent if the call queues to the same split with a different priority.

This event report is not generated when a call queues to an announcement, Vector announcement or trunk group. It is also not generated when a call queues, again, to the same ACD split at the same priority.

Refer to the Detailed Information: section below for specific instructions to program your application to obtain this event.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
queuedConnection	[mandatory] Specifies the connection that queued.
queue	[mandatory] Specifies the queuing device to which the call has queued. This is the extension of the ACD split to which the call queued.
callingDevice	[mandatory - partially supported] Specifies the calling device. The deviceIDStatus may be ID_NOT_KNOWN.
calledDevice	[mandatory - partially supported] Specifies the called device. The following rules apply: For incoming calls over PRI facilities, the "called number" from the ISDN SETUP message is specified. If the "called number" does not exist (i.e., NULL), the deviceIDStatus is ID_NOT_KNOWN. For incoming calls over non-PRI facilities the called number is the principal extension (a group extension for TEG, PCOL, hunt group, VDN). If the switch is administered to modify the DNIS digits, then the modified DNIS is specified. For outbound calls, dialed number is specified.
lastRedirectionDevice	[optional - limited support] Specifies the previous redirection/alerted device in case where the call was redirected/diverted to the queue device.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
numberQueued	[optional - supported] Specifies how many calls are queued to the queue device. This is the call position in the queue in the hunt group or ACD split. This number will include the current call and excludes all direct-agent calls in the queue.
cause	[optional - supported] Specifies the cause for this event. The following cause is supported: ● EC_REDIRECTED - The call has been redirected. ● EC_SINGLE_STEP_TRANSFER (private data version 8 or later) - The call was queued as the result of a Single Step Transfer Call operation.

Private Parameters

deviceHistory

The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the deviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

The deviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the deviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the deviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if a ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Detailed Information:

In addition to the information provided below, see the Event Report Detailed Information section in this chapter.

- Last Redirection Device - There is only limited support for this parameter. An application must understand the limitations of this parameter in order to use the information correctly.

Perform either of the steps below to obtain the queued event in your application with a Definity ECS:

- For any vector controlled ACD or Skill (EAS or Non-EAS) use cstaMonitorCallViaDevice() to monitor the VDN that queues calls to the ACD or Skill.
- For a non-vector controlled ACD (Non-EAS) use cstaMonitorcallsViaDevice() to monitor the device.

Syntax

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

// CSTAQueuedEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_QUEUED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
                  CSTAQueuedEvent_t  queued;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_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;
```

Private Data Version 7 and 8 Syntax

The CSTA Queued Event includes a private data event, ATTQueuedEvent for private data version 7. The ATTQueuedEvent uses the deviceHistory private data parameter.

The deviceHistory parameter is new for private data version 7.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTQueuedEvent - CSTA Unsolicited Event Private Data

typedef struct ATTQueuedEvent_t {
    DeviceHistory_t deviceHistory;
} ATTQueuedEvent_t;
```

Retrieved Event

Summary

- Direction: Switch to Client
- Event: CSTARetrievedEvent
- Service Parameters: monitorCrossRefID, retrievedConnection, retrievingDevice, localConnectionInfo, cause

Functional Description:

The Retrieved Event Report indicates that the switch detects a previously held call that has been retrieved.

Before **After**

It is generated when an on-PBX station connects to a call that has been previously placed on hold. Retrieving to a held call can be done either manually at the station by selecting the call appearance of the held call or by switch-hook flash from an analog station, or via a cstaRetrieveCall Service request from a client application.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
retrievedConnection	[mandatory] Specifies the connection for which the call has been taken off the hold state.
retrievingDevice	[mandatory] Specifies the device that connected the call from the hold state. This is the extension that has been connected the call.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - supported] Specifies the cause for this event. The following cause is supported: EC_KEY_CONFERENCE - Indicates that the event report occurred at a bridged device.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

// CSTARetrievedEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_RETRIEVED
      } ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_t monitorCrossRefId;
              union
              {
                  CSTARetrievedEvent_t  retrieved;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

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

Service Initiated Event

Summary

- Direction: Switch to Client
- Event: CSTAServiceInitiatedEvent
- Private Data Event: ATTServiceInitiatedEvent
- Service Parameters: monitorCrossRefID, initiatedConnection, localConnectionInfo, cause
- Private Parameters: ucid

Functional Description:

The Service Initiated Event Report indicates that telecommunication service is initiated.

Before **After**

This event is generated as follows:

- When a station begins to receive dial tone.
- When a station is forced off-hook because a cstaMakeCall is requested on that station.
- When certain switch features that initiate a call (such as the abbreviated dialing, etc.) are invoked.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
initiatedConnection	[mandatory] Specifies the connection for which the service (dial tone) has been initiated.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for the cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - supported] Specifies the cause for this event. The following cause is supported: • EC_KEY_CONFERENCE - Indicates that the event report occurred at a bridged device.

Private Parameters:

ucid	[optional] Specifies the Universal Call ID (UCID) of the resulting call. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Detailed Information:

See the Event Report Detailed Information section in this chapter.

Syntax

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

// CSTAServiceInitiatedEvent

typedef struct
{
      ACSHandle_t acsHandle;
      EventClass_teventClass; // CSTAUNSOLICITED
      EventType_t eventType;  // CSTA_SERVICE_INITIATED
} ACSEventHeader_t;

typedef struct
{
      ACSEventHeader_teventHeader;
      union
      {
          struct
          {
              CSTAMonitorCrossRefID_tmonitorCrossRefId;
              union
              {
              CSTAServiceInitiatedEvent_t  serviceInitiated;
              } u;
          } cstaUnsolicited;
      } event;
      charheap[CSTA_MAX_HEAP];
} CSTAEvent_t;

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

Private Parameter Syntax

If private data accompanies a CSTAServiceInitiatedEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTAServiceInitiatedEvent does not deliver private data to the application. If the acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
 #include <attpriv.h>

 // ATTServiceInitiatedEvent - CSTA Unsolicited Event Private Data
// (supported by private data version 5 and later only)

 typedef struct
 {
      ATTEventTypeeventType;// ATT_SERVICE_INITIATED
      union
      {
          ATTServiceInitiatedEvent_tserviceInitiated;
      } u;
 } ATTEvent_t;

 typedef struct ATTServiceInitiatedEvent_t
 {
      ATTUCID_t   ucid;
 } ATTServiceInitiatedEvent_t;

 typedef char ATTUCID_t[64];
```

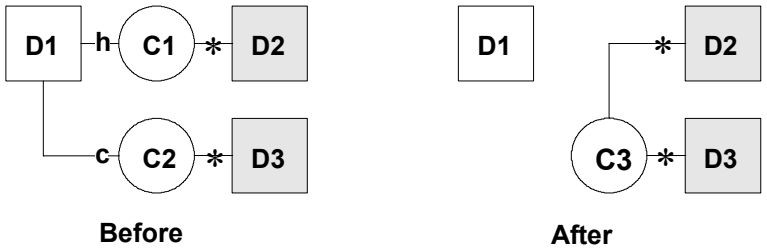
Transferred Event

Summary

- Direction: Switch to Client
- Event: CSTATransferredEvent
- Private Data Event: ATTTransferredEvent (private data version 7), ATTV6TransferredEvent (private data version 6), ATTV5TransferredEvent (private data version 5), ATTV4TransferredEvent (private data version 4), ATTV3TransferredEvent (private data versions 2 and 3)
- Service Parameters: *monitorCrossRefID, primaryOldCall, secondaryOldCall, transferringDevice, transferredDevice, transferredConnections, localConnectionInfo, cause*
- Private Parameters: *originalCallInfo, distributingDevice, distributingVDN, ucid, trunkList, deviceHistory*

Functional Description:

The Transferred Event Report indicates that an existing call was transferred to another device and the device requesting the transfer has been dropped from the call. The transferringDevice will not appear in any future feedback for the call.



The Transferred Event Report is generated for the following circumstances:

- When an on-PBX station completes a transfer by pressing the "transfer" button on the voice terminal.
- When the on-PBX analog set (phone) user on a monitored call goes on hook with one active call and one call on conference/transfer hold.
- When the "call park" feature is used in conjunction with the "transfer" button on the voice set.
- When an adjunct successfully completes a cstaTransferCall request.

Service Parameters:

monitorCrossRefID	[mandatory] Contains the handle to the monitor request for which this event is reported.
primaryOldCall	[mandatory] Specifies the callID of the call that was transferred. This is usually the held call before the transfer. This call ended as a result of the transfer.
secondaryOldCall	[mandatory] Specifies the callID of the call that was transferred. This is usually the active call before the transfer. This call is retained by the switch after the transfer.
transferringDevice	[mandatory] Specifies the device that is controlling the transfer. This is the device that did the transfer.
transferredDevice	[mandatory] Specifies the new transferred-to device. If the device is an on-PBX station, the extension is specified. If the party is an off-PBX endpoint, then the deviceIDStatus is ID_NOT_KNOWN. There are call scenarios in which the transfer operation joins multiple parties to a call. In such situations, the transferredDevice will be the extension for the last party to join the call.
transferredConnections	[optional - supported] Specifies a count of the number of devices and a list of connectionIDs and deviceIDs which resulted from the transfer. If a device is on-PBX, the extension is specified. The extension consists of station or group extensions. Group extensions are provided when the transfer is to a group and the transfer completes before the call is answered by one of the group members (TEG, PCOL, hunt group, or VDN extension). It may contain alerting extensions. The static deviceID of a queued endpoint is set to the split extension of the queue. If a party is off-PBX, then its static device identifier or its previously assigned trunk identifier is specified.
localConnectionInfo	[optional - supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for the cstaMonitorDevice requests only. A value of CS_NONE indicates that the local connection state is unknown.
cause	[optional - supported] Specifies the cause for this event. The following causes are supported: ● EC_TRANSFER - A call transfer has occurred. ● EC_PARK - A call transfer was performed for parking a call rather than a true call transfer operation. ● EC_SINGLE_STEP_TRANSFER (private data version 8 or later) - The call was transferred via the Single Step Transfer Call service.

Private Parameters:

originalCallInfo [optional] Specifies the original call information. This parameter is sent with this event for the resulting newCall of a cstaTransferCall request or the retained call of a (manual) transfer call operation. The calls being transferred must be known to the TSAPI Service via the Call Control Services or Monitor Services.

For a cstaTransferCall, the originalCallInfo includes the call information originally received by the heldCall specified in the cstaTransferCall request. For a manual call transfer, the originalCallInfo includes the call information originally received by the primaryOldCall specified in the event report.

- ***reason*** - the reason for the originalCallInfo. The following reasons are supported.
  - OR_NONE - no originalCallInfo provided
  - OR_CONFERENCED - call conferenced
  - OR_CONSULTATION - consultation call
  - OR_TRANSFERRED - call transferred
  - OR_NEW_CALL - new call
- ***callingDevice*** - The original callingDevice received by the heldCall or the primaryOldCall. This parameter is always provided.
- ***calledDevice*** - The original calledDevice received by the heldCall or the primaryOldCall. This parameter is always provided.
- ***trunk*** - The original trunk group received by the heldCall or the primaryOldCall. This parameter is supported by private data versions 2, 3, and 4.
- ***trunkGroup*** - The original trunk group received by the heldCall or the primaryOldCall. This parameter is supported by private data version 5 and later only.
- ***trunkMember*** (G3V4 switches and later) - The original trunkMember received by the heldCall or the primaryOldCall.
- ***lookaheadInfo*** - The original lookaheadInfo received by the heldCall or the primaryOldCall.
- ***userEnteredCode*** - The original userEnteredCode received by the heldCall or the primaryOldCall call.
- ***userInfo*** - the original userInfo received by the heldCall or the primaryOldCall call.
- Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo is increased to 96 bytes.
- An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.
- ***ucid*** - the original ucid of the call. This parameter is supported by private data version 5 and later only.

originalCallInfo (continued)

- ***callOriginatorInfo*** - the original callOriginatorInfo received by the call. This parameter is supported by private data version 5 and later only.
- ***flexibleBilling*** - the original flexibleBilling information of the call. This parameter is supported by private data version 5 and later only.
- ***deviceHistory*** - The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. For an explanation of this parameter and the following list of entries, see deviceHistory on page 661.
  - **oldDeviceID (M) DeviceID**
  - **EventCause (O) EventCause**
  - **OldConnectionID (O) ConnectionID**

See the Delivered Event section in this chapter for details on these parameters.

distributingDevice [optional] specifies the original distributing device before the call is transferred. See the Delivered Event section in this chapter for details on the distributingDevice parameter. This parameter is supported by private data version 4 and later.

distributingVDN The VDN extension associated with the distributing device. The field gets set only and exactly under the following conditions.

- When the application monitors the VDN in question and sees the C_OFFERED (translated potentially into a Delivered event, if the application does not filter it out)
- When the application monitors an agent and receives a call that came from that monitored VDN (that is, in the Delivered, Established, Transferred, and Conferenced events).

ucid [optional] Specifies the Universal Call ID (UCID) of the call. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

trunkList [optional] Specifies a list of up to 5 trunk groups and trunk members. This parameter is supported by private data version 6 and later only. The following options are supported:

- count - The count of the connected parties on the call.
- trunks - An array of 5 trunk group and trunk member IDs, one for each connected party. The following options are supported:
  - connection - The connection ID of one of the parties on the call.
  - trunkGroup - The trunk group of the party referenced by connection.
  - trunkMember - The trunk member of the party referenced by connection.

deviceHistory

The deviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the deviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

The deviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the deviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the deviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if an ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Detailed Information:

In addition to the information provided below, see the Event Report Detailed Information section in this chapter.

The originalCallInfo includes the original call information originally received by the call that is ended as the result of the transfer. The following special rules apply:

- If the Transferred Event was a result of a cstaTransferCall request, the originalCallInfo and the distributingDevice sent with this Transferred Event is from the heldCall in the cstaTransferCall request. Thus the application can control the originalCallInfo and the distributingDevice to be sent in a Transferred Event by putting the original call on hold and specifying it as the heldCall in the cstaTransferCall request. Although the primaryOldCall that is the call ended as the result of the cstaTransferCall is the heldCall most of the time, sometimes it can be the activeCall.
- If the Transferred Event was a result of a manual transfer, the originalCallInfo and the distributingDevice sent with this Transferred Event is from the primaryOldCall of the event. Thus the application does not have control of the originalCallInfo and distributingDevice to be sent in the Transferred Event. Although the primaryOldCall that is the call ended as the result of the manual transfer operation is the heldCall most of the time, sometimes it can be the active call.

In addition, see the Established Event Detailed Information: section for Blind Transfer and Consultation Transfer definitions; Transferring or conferencing a call together with screen pop information on page 45 for the recommended design for applications that use caller information to populate a screen); and the ANI Screen Pop Application Requirements in the Event Report Detailed Information section in this chapter.

Syntax

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

 // CSTATransferredEvent

 typedef struct
 {
      ACSHandle_t  acsHandle;
      EventClass_t eventClass;  // CSTAUNSOLICITED
      EventType_t  eventType;   // CSTA_TRANSFERRED
 } ACSEventHeader_t;

 typedef struct
 {
      ACSEventHeader_t  eventHeader;
      union
      {
           struct
           {
                CSTAMonitorCrossRefID_t    monitorCrossRefId;
                union
                {
                     CSTATransferredEvent_t  transferred;
                } u;
           } cstaUnsolicited;
      } event;
      char heap[CSTA_MAX_HEAP];
 } CSTAEvent_t;

 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;

 typedefExtendedDeviceID_tSubjectDeviceID_t;

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

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

Private Data Version 7 and 8 Syntax

If private data accompanies a CSTATransferredEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTATransferredEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

The deviceHistory parameter is new for private data version 7.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

 // ATTTransferredEvent - CSTA Unsolicited Event Private Data

typedef struct ATTTransferredEvent_t {
    ATTOriginalCallInfo_t originalCallInfo;
    CalledDeviceID_t distributingDevice;
    ATTUCID_t       ucid;
    ATTTrunkList_t  trunkList;
    DeviceHistory_t deviceHistory;
    CalledDeviceID_t distributingVDN;
} ATTTransferredEvent_t;
```

Private Data Version 6 Syntax

If private data accompanies a CSTATransferredEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTATransferredEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

 // ATTV6TransferredEvent - CSTA Unsolicited Event Private Data

 typedef struct
 {
      ATTEventType_teventType;// ATTV6_TRANSFERRED
      union
      {
          ATTV6TransferredEvent_t v6transferredEvent;
      } u;
 } ATTEvent_t;

 typedef struct ATTTransferredEvent_t
 {
      ATTOriginalCallInfo_t   originalCallInfo;
      CalledDeviceID_t        distributingDevice;
      ATTUCID_t               ucid;
      ATTTrunkList_t          trunkList;
} ATTTransferredEvent_t;

 typedef struct ATTOriginalCallInfo_t
 {
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunkGroup;
      DeviceID_t              trunkMember;
      ATTLookaheadInfo_t      lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTUserToUserInfo_t     userInfo;
      ATTUCID_t               ucid;
      ATTCallOriginatorInfo_t callOriginatorInfo;
      Boolean                 flexibleBilling;
 } ATTOriginalCallInfo_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef enum ATTReasonForCallInfo_t
 {
      OR_NONE         = 0,// indicates info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
 } ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;

typedef struct ATTLookaheadInfo_t
{
      ATTInterflow_t      type;
      ATTPriority_t       priority;
      short               hours;
      short               minutes;
      short               seconds;
      DeviceID_t          sourceVDN;
      ATTUnicodeDeviceID_tuSourceVDN;
                          // sourceVDN in Unicode
 } ATTLookaheadInfo_t;

 typedef enum ATTInterflow_t
 {
      LAI_NO_INTERFLOW= -1,// indicates info not present
      LAI_ALL_INTERFLOW= 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
 } ATTInterflow_t;

 typedef enum ATTPriority_t
 {
      LAI_NOT_IN_QUEUE= 0,
      LAI_LOW         = 1,
      LAI_MEDIUM      = 2,
      LAI_HIGH        = 3,
      LAI_TOP         = 4
 } ATTPriority_t;

typedef struct ATTUnicodeDeviceID_t
 {
      short           count;
      unsigned short  value[64];
 } ATTUnicodeDeviceID_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t      type;
      ATTUserEnteredCodeIndicator_t indicator;
      char                          data[ATT_MAX_USER_CODE];
      DeviceID_t                    collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE          = -1,// indicates not specified
      UE_ANY           = 0,
      UE_LOGIN_DIGITS  = 2,
      UE_CALL_PROMPTER = 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT= 0,
      UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTUserToUserInfo_t
{
      ATTUUIProtocolType_t type;
      struct {
          short            length;
                           // 0 indicates UUI not present
          unsigned char    value[ATT_MAX_USER_INFO];
      } data;
 } ATTUserToUserInfo_t;

 typedef enum ATTUUIProtocolType_t
 {
      UUI_NONE             = -1,// indicates not specified
      UUI_USER_SPECIFIC    = 0,// user-specific
      UUI_IA5_ASCII        = 4
                  // null terminated ascii character string
} ATTUUIProtocolType_t;

typedef char ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t
{
      Boolean  hasInfo;// if FALSE, no callOriginatorType
      short    callOriginatorType;
 } ATTCallOriginatorInfo_t;
```

Private Data Version 5 Syntax

If private data accompanies a CSTATransferredEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTATransferredEvent does not deliver private data to the application. If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

 // ATTV5TransferredEvent - CSTA Unsolicited Event Private Data

 typedef struct
 {
      ATTEventType_teventType;// ATT_TRANSFERRED
      union
      {
          ATTV5TransferredEvent_ttransferredEvent;
      } u;
 } ATTEvent_t;

 typedef struct ATTV5TransferredEvent_t
 {
      ATTV5OriginalCallInfo_t originalCallInfo;
      CalledDeviceID_t        distributingDevice;
 ATTUCID_t                    ucid;
 } ATTV5TransferredEvent_t;

 typedef struct ATTV5OriginalCallInfo_t
 {
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunkGroup;
      DeviceID_t              trunkMember;
      ATTLookaheadInfo_t      lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTV5UserToUserInfo_t   userInfo;
      ATTUCID_t               ucid;
      ATTCallOriginatorInfo_t callOriginatorInfo;
      Boolean                 flexibleBilling;
 } ATTV5OriginalCallInfo_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef enum ATTReasonForCallInfo_t
 {
      OR_NONE         = 0,// indicates info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
 } ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;

typedef struct ATTLookaheadInfo_t
{
      ATTInterflow_t      type;
      ATTPriority_t       priority;
      short               hours;
      short               minutes;
      short               seconds;
      DeviceID_t          sourceVDN;
      ATTUnicodeDeviceID_tuSourceVDN;
                          // sourceVDN in Unicode
 } ATTLookaheadInfo_t;

 typedef enum ATTInterflow_t
 {
      LAI_NO_INTERFLOW= -1,// indicates info not present
      LAI_ALL_INTERFLOW= 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
 } ATTInterflow_t;

 typedef enum ATTPriority_t
 {
      LAI_NOT_IN_QUEUE= 0,
      LAI_LOW         = 1,
      LAI_MEDIUM      = 2,
      LAI_HIGH        = 3,
      LAI_TOP         = 4
 } ATTPriority_t;

typedef struct ATTUnicodeDeviceID_t
 {
      short           count;
      unsigned short  value[64];
 } ATTUnicodeDeviceID_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t      type;
      ATTUserEnteredCodeIndicator_t indicator;
      char                          data[ATT_MAX_USER_CODE];
      DeviceID_t                    collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE          = -1,// indicates not specified
      UE_ANY           = 0,
      UE_LOGIN_DIGITS  = 2,
      UE_CALL_PROMPTER = 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT= 0,
      UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_t type;
      struct {
          short            length;
                           // 0 indicates UUI not present
          unsigned char    value[33];
      } data;
 } ATTV5UserToUserInfo_t;

 typedef enum ATTUUIProtocolType_t
 {
      UUI_NONE             = -1,// indicates not specified
      UUI_USER_SPECIFIC    = 0,// user-specific
      UUI_IA5_ASCII        = 4
                  // null terminated ascii character string
} ATTUUIProtocolType_t;

typedef char ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t
{
      Boolean  hasInfo;// if FALSE, no callOriginatorType
      short    callOriginatorType;
 } ATTCallOriginatorInfo_t;
```

Private Data Version 4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

 // ATTV4TransferredEvent - CSTA Unsolicited Event Private Data

 typedef struct
 {
      ATTEventType_teventType;// ATTV4_TRANSFERRED
      union
      {
          ATTV4TransferredEvent_tv4transferredEvent;
      } u;
 } ATTEvent_t;

 typedef struct ATTV4TransferredEvent_t
 {
      ATTV4OriginalCallInfo_t originalCallInfo;
      CalledDeviceID_t        distributingDevice;
 } ATTV4TransferredEvent_t;

 typedef struct ATTV4OriginalCallInfo_t
 {
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunk;
      DeviceID_t              trunkMember;
      ATTV4LookaheadInfo_t    lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTV5UserToUserInfo_t   userInfo;
 } ATTV4OriginalCallInfo_t;

 typedef enum ATTReasonForCallInfo_t
 {
      OR_NONE         = 0,// indicates info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
} ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;
```

Private Data Version 4 Syntax (Continued)

```
typedef struct ATTV4LookaheadInfo_t
{
      ATTInterflow_t   type;
      ATTPriority_t    priority;
      short            hours;
      short            minutes;
      short            seconds;
      DeviceID_t       sourceVDN;
 } ATTV4LookaheadInfo_t;

 typedef enum ATTInterflow_t
 {
      LAI_NO_INTERFLOW= -1,// indicates info not present
      LAI_ALL_INTERFLOW= 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
 } ATTInterflow_t;

 typedef enum ATTPriority_t
 {
      LAI_NOT_IN_QUEUE= 0,
      LAI_LOW          = 1,
      LAI_MEDIUM       = 2,
      LAI_HIGH         = 3,
      LAI_TOP          = 4
 } ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t      type;
      ATTUserEnteredCodeIndicator_tindicator;
      char                          data[ATT_MAX_USER_CODE];
      DeviceID_t                    collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE          = -1,// indicates not specified
      UE_ANY           = 0,
      UE_LOGIN_DIGITS  = 2,
      UE_CALL_PROMPTER = 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT= 0,
      UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;
```

Private Data Version 4 Syntax (Continued)

```
typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_t type;
      struct {
          short   length;    // 0 indicates UUI not present
          unsigned charvalue[33];
      } data;
 } ATTV5UserToUserInfo_t;

 typedef enum ATTUUIProtocolType_t
 {
      UUI_NONE        = -1,// indicates not specified
      UUI_USER_SPECIFIC= 0,// user-specific
      UUI_IA5_ASCII= 4    // null terminated ascii
                          // character string
 } ATTUUIProtocolType_t;
```

Private Data Versions 2 and 3 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

 // ATTV3TransferredEvent - CSTA Unsolicited Event Private Data

 typedef struct
 {
      ATTEventType_teventType;// ATTV3_TRANSFERRED
      union
      {
          ATTV3TransferredEvent_tv3transferredEvent;
      } u;
 } ATTEvent_t;

 typedef struct ATTV3TransferredEvent_t
 {
      ATTV4OriginalCallInfo_toriginalCallInfo;
 } ATTV3TransferredEvent_t;

 typedef struct ATTV4OriginalCallInfo_t
 {
      ATTReasonForCallInfo_t  reason;
      CallingDeviceID_t       callingDevice;
      CalledDeviceID_t        calledDevice;
      DeviceID_t              trunk;
      DeviceID_t              trunkMember;
      ATTV4LookaheadInfo_t    lookaheadInfo;
      ATTUserEnteredCode_t    userEnteredCode;
      ATTV5UserToUserInfo_t   userInfo;
 } ATTV4OriginalCallInfo_t;

 typedef enum ATTReasonForCallInfo_t
 {
      OR_NONE         = 0,// indicates info not present
      OR_CONSULTATION = 1,
      OR_CONFERENCED  = 2,
      OR_TRANSFERRED  = 3,
      OR_NEW_CALL     = 4
 } ATTReasonForCallInfo_t;

typedef ExtendedDeviceID_tCallingDeviceID_t;

typedef ExtendedDeviceID_tCalledDeviceID_t;
```

Private Data Versions 2 and 3 Syntax (Continued)

```
typedef struct ATTV4LookaheadInfo_t
{
      ATTInterflow_t   type;
      ATTPriority_t    priority;
      short            hours;
      short            minutes;
      short            seconds;
      DeviceID_t       sourceVDN;
 } ATTV4LookaheadInfo_t;

 typedef enum ATTInterflow_t
 {
      LAI_NO_INTERFLOW= -1,// indicates info not present
      LAI_ALL_INTERFLOW= 0,
      LAI_THRESHOLD_INTERFLOW= 1,
      LAI_VECTORING_INTERFLOW= 2
 } ATTInterflow_t;

 typedef enum ATTPriority_t
 {
      LAI_NOT_IN_QUEUE = 0,
      LAI_LOW          = 1,
      LAI_MEDIUM       = 2,
      LAI_HIGH         = 3,
      LAI_TOP          = 4
 } ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
      ATTUserEnteredCodeType_t      type;
      ATTUserEnteredCodeIndicator_tindicator;
      char                          data[ATT_MAX_USER_CODE];
      DeviceID_t                    collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
      UE_NONE          = -1,// indicates not specified
      UE_ANY           = 0,
      UE_LOGIN_DIGITS  = 2,
      UE_CALL_PROMPTER = 5,
      UE_DATA_BASE_PROVIDED= 17,
      UE_TONE_DETECTOR= 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
      UE_COLLECT= 0,
      UE_ENTERED= 1
} ATTUserEnteredCodeIndicator_t;
```

Private Data Versions 2 and 3 Syntax (Continued)

```
typedef struct ATTV5UserToUserInfo_t
{
      ATTUUIProtocolType_t type;
      struct {
          short   length;    // 0 indicates UUI not present
          unsigned charvalue[33];
      } data;
 } ATTV5UserToUserInfo_t;

 typedef enum ATTUUIProtocolType_t
 {
      UUI_NONE            = -1,// indicates not specified
      UUI_USER_SPECIFIC   = 0,// user-specific
      UUI_IA5_ASCII       = 4 // null terminated ascii
                              // character string
 } ATTUUIProtocolType_t;
```

Event Report Detailed Information

Analog Sets

Redirection

Analog sets do not support temporary bridged appearances. When, in normal circumstances, a call at a multifunction set would have been left on a simulated bridge appearance, the call will move away from the analog set. Thus, any monitor requests for the analog set will receive the Diverted Event Report.

Delivered Event Reports are not sent to SAC-activated analog sets receiving calls.

Redirection on No Answer

Calls redirected by this feature generate the following event reports when a call is redirected from a nonanswering station.

- Diverted Event Report is provided over the cstaMonitorDevice monitor requests when the call is redirected from a nonanswering agent. This event is not provided if the call is queued again to the split or delivered to another agent in the split.
- Queued Event Report will be generated if the call queues after being redirected.
- Call Cleared Event Report - If the call cannot re-queue after the call has been redirected from the nonanswering agent, then the call continues to listen to ringback until the caller is dropped. In this case, a Call Cleared Event Report is generated when the caller is dropped and the call disconnected.

Direct Agent Calls always redirect to the agent's coverage path instead of queueing again to the servicing ACD split.

Switch Hook Operation

When an analog set goes on-hook with one or two calls on hold, the user is audibly notified (the phone rings). This notification ring is not reported as a Delivered event. When the user goes off-hook and is reconnected to the alerting call, a Retrieved Event Report is generated.

When a user goes on hook with a soft-held call and an active call, both calls are transferred away from the user's set. It does not matter how the held call was placed on soft hold.

If a monitored analog user flashes the switch hook to put a call on soft hold to start a new call:

- The Held Event Report is sent to all monitor requests.

- A Service Initiated Event Report is returned to all cstaMonitorDevice requests when the user receives the dial tone.
- A Retrieved Event Report is returned to all monitor requests if the user returns to the held call. If the held call is conferenced or transferred, the Conferenced or Transferred Event Reports are sent to all monitor requests.

ANI Screen Pop Application Requirements

The list below summarizes the prerequisites for ANI screen pop at a station. Each item is discussed in more detail below:

- Incoming PRI provides ANI for incoming external calls. No other external sources (such as "caller-id") are supported. This is a typical G3 call center configuration.
- Local Communication Manager server or DCS provides extension number as ANI for local or private network incoming calls. This is a typical help desk configuration.
- Chapter 3 gives design guidelines for transferring a call across more than one TSAPI Service servers, across CTI platforms, and across switches. If these guidelines are not followed, then the transferring party and receiving party must be on the same switch and monitored by the same TSAPI Service. Transfers across a private DCS network are not supported.
- The receiving party may either manually answer the call or run an application that uses cstaAnswerCall.

If the design considerations in Chapter 3 are not followed, then ANI screen pop on blind transfer can only be done at the time the call is answered, not when it rings. In this case, applications will find the ANI information in the CSTA Established Event (which the driver sends when the call answers), not the CSTA Delivered Event (which the driver sends when the call rings). For an application to do an ANI screen pop on a blind transfer, it must look in the proper CSTA Event.

If the design considerations in Chapter 3 are not followed, then ANI screen pop on consultation transfer is possible only at the time the call transfers, not when the consultation call rings or is answered. In this case, applications will find the information necessary to do the screen pop in the CSTA Transfer Event (which the driver sends them when the call transfers), not in the CSTA Established or Delivered events. For an application to do an ANI screen pop on a consultation transfer, it must look in the proper CSTA Event.

If the design considerations in Chapter 3 are not followed, then ANI screen pop on a consultation transfer requires that the transferring party must be monitored by the same TSAPI Service that is monitoring the receiving party.

Announcements

Automatic Call Distribution (ACD) split-forced announcements and vector announcements do not generate event reports for the application. However, nonsplit announcements generate events that are sent to other parties on the call.

Extensions assigned to integrated announcements may not be monitored.

Answer Supervision

The Communication Manager "answer supervision timeout" field determines how long the central office trunk board waits before sending the (simulated) "answer" message to the software. This is useful when the answer supervision is not available on a trunk. This message is used to send call information to Station Message Detail Recording (SMDR) and to trigger the bridging of a service observer onto an outgoing trunk call. This message is ignored if the trunk is expected to receive true answer supervision from the network (the switch uses the true answer supervision whenever available). Client application monitored calls are treated like regular calls. No Established Event Report will be generated for this "simulated answer."

With respect to cstaMakePredictiveCall calls, when the "answer supervision"' field is set to "no", the switch relies entirely on the call classifier to determine when the call was answered. When answer supervision on the trunk is set to "yes", a cstaMakePredictiveCall call is considered "answered" when the switch software receives the "answer" message from the trunk board. In reality, cstaMakePredictiveCall calls may receive either an "answer" message from the trunk board or (if this never comes) an indication from the classifier that the far end answered. In this case, the switch will act on the first indication received and not act on any subsequent indications.

Attendants and Attendant Groups

An attendant group extension cannot be monitored as a station.

Individual attendants may be parties on monitored calls and are supported like regular station users as far as the event reporting is concerned on monitors for other station types.

An attendant group may be a party on a monitored call, but the Delivered, Established, and Connection Cleared Event Reports do not apply.

An individual attendant extension member cannot be monitored by a cstaMonitorDevice request; but it can be a destination for a call from a cstaMonitorDevice monitored station. In this case, event reports are sent to the cstaMonitorDevice request about the individual attendant that is receiving the call.

Attendant Specific Button Operation

This section clarifies what events are sent when an attendant uses buttons that are specific to an attendant console.

- Hold button - If an individual attendant presses the hold button and the call is monitored, the Held Event Report will be sent to the corresponding monitor request.
- Call Appearance button - If an individual attendant has a call on hold, and the call is monitored, then the Retrieved Event Report will be sent to the corresponding monitor requests.
- Start button - If a call is present at an attendant and the call is monitored, and the attendant presses the start button, then the call will be put on hold and a Held Event Report will be sent on the corresponding monitor requests.
- Cancel button - If a call is on hold at the attendant and the attendant presses the start button, putting the previous call on hold, and then either dials a number and then presses the cancel button or presses the cancel button right away, the call that was originally put on hold will be reconnected and a Retrieved Event Report will be sent to the monitor request on the call.
- Release button - If only one call is active and the attendant presses the release button, the call will be dropped and the Connection Cleared Event Report will be sent to the monitor request on the call. If two calls are active at the attendant and the attendant then presses the release button, the calls will be transferred away from the attendant and a Transferred Event Report will be sent to the monitor request on the calls.
- Split button - If two calls are active at the attendant and the attendant presses the split button, the calls will be conferenced at the attendant and a Conferenced Event Report will be sent to the monitor requests monitoring the calls.

Attendant Auto-Manual Splitting

If an individual attendant receives a call with cstaMonitorDevice requests, then activates the Attendant Auto-Manual Splitting feature, a Held Event Report is returned to the monitor requests. The next event report sent depends on which button the attendant presses on the set (CANCEL = Retrieved, SPLIT = Conferenced, RELEASE = Transferred).

Attendant Call Waiting

Calls that provide event reports over cstaMonitorDevice requests and are extended by an attendant to a local, busy, single-line voice terminal will generate the following event reports:

- Held when the incoming call is split away by the attendant.

- Established when the attendant returns to the call.

The following events are generated, if the busy station does not accept the extended call and its returns:

- Delivered when the call is returned to the attendant.
- Established when the attendant returns to the call.

Attendant Control of Trunk Group Access

Calls that provide event reports over cstaMonitorDevice requests can access any trunk group controlled by the attendant. The attendant is alerted and places the call to its destination.

AUDIX

Calls that cover AUDIX do not maintain a simulated bridge appearance on the principal's station. The principal receives audible alerting followed by an interval of coverage response followed by the call dropping from the principal's set. When the principal receives alerting, the Delivered Event Report is sent. When the call is dropped from the principal's set because the call went to AUDIX coverage, the Diverted Event Report is sent.

Automatic Call Distribution (ACD)

Announcements

Announcements played while a monitored call is in a split queue, or as a result of an announcement vector command, create no event reports. Calls made directly to announcement extensions will have the same event report sent to the application as calls made to station extensions. In either case, no Queued Event Report is sent to the application.

Interflow

This occurs when a split redirects all calls to another split on another PBX by activating off-premise call forwarding.

When a monitored call interflows, event reports will cease except for the Network Reached (for non-PRI trunk) and trunk Connection Cleared Event Reports.

Night Service

The Delivered Event Report is sent when a call that is not being monitored enters an ACD split (not adjunct-controlled) with monitor requests and also has night service active.

Service Observing

A monitored call can be service observed provided that service observing is originated from a voice terminal and the service observing criteria is met. An Established Event Report is generated every time service observing is activated for a monitored call. A Connection Cleared Event Report is generated when the observer disconnects from the call.

For a cstaMakeCall call, the observer is bridged on the connection when the destination answers. When the destination is a trunk with answer supervision (includes PRI), the observer is bridged on when an actual far-end answer occurs. When the destination is a trunk without answer supervision, the observer is bridged on after the Network Reached (timeout) event.

Applicable events are "Established" (when the observer is bridged on) with the observer's extension and "Connection Cleared" when the observer drops from the call. In addition, the observer may manipulate the call via Call Control requests to the same extent as he or she can via the station set.

Auto-Available Split

An auto-available split can be monitored as an ACD split and members of auto-available splits (agents) can be monitored as stations.

Bridged Call Appearance

A cstaMonitorDevice monitored station can have a bridged appearance(s) of its primary extension number appear at other stations. For bridging, event reports are provided based on the internal state of bridging parties with respect to the call. A call to the primary extension number will alert both the principal and the bridged appearance. Two or more Delivered Event Reports get triggered, one for the principal, and one for each of the bridged appearances. Two or more Established Event Reports may be triggered, if both the primary extension number and the bridged appearance(s) pick up the call. When the principal or bridging user goes on hook but the bridge itself does not drop from the call, no event report is sent but the state of that party changes from the connected state to the bridged state. When the principal or bridging user reconnects, another Established Event Report will be sent. A Connection Cleared Event Report will be triggered for the principal and each bridged appearance when the entire bridge drops from the call.

Members that are not connected to the call while the call is connected to another bridge member are in the "bridged" state. When the only connected member of the bridge transitions to the held state, the state for all members of the bridge changes to the held state even if they were previously in the bridged state. There is no event report sent to the bridged user monitor request for this transition.

Both the principal and bridging users may be individually monitored by a cstaMonitorDevice. Each will receive appropriate events as applicable to the monitored station. However, event reporting for a member of the bridge in the held state will be dependent on whether the transition was from the connected state or the bridged state.

CSTA Conference Call, Drop Call, Hold Call, Retrieve Call, and Transfer Call services are not permitted for parties in the bridged state and may also be more restrictive if the principal of the bridge has an analog set or if the exclusion option is in effect from a station associated with the bridge.

A CSTA Make Call request will always originate at the primary extension number of a user having a bridged appearance. For a call to originate at the bridged call appearance of a primary extension, that user must be off hook at that bridged appearance at the time the request is received.

Note:

A principal station with bridged call appearance can be single step conferenced into a call. Stations with bridged call appearance to the principal have the same bridged call appearance behavior, that is, if monitored, the station will receive Established And Conferenced Events when it joins the call. The station will not receive a Delivered Event.

Busy Verification of Terminals

A cstaMonitorDevice-monitored station may be busy-verified. An Established Event Report is provided when the verifying user is bridged in on a connection in which there is a cstaMonitorDevice-monitored station.

Call Coverage

If a call that goes to coverage is monitored by a monitor request on an ACD split or a VDN, the monitor request will receive the Delivered and Established Event reports.

For an alternate answering position that is monitored by a cstaMonitorDevice request, the Delivered and Established Event Reports are returned to its cstaMonitorDevice request.

The Diverted Event Report is sent to the principal’s cstaMonitorDevice request when an analog principal’s call goes to coverage. The Connection Cleared Event Report is sent for the coverage station’s monitor requests when the call that had been alerting at both the principal and the coverage is picked up at the principal.

Call Coverage Path Containing VDNs

When a call is diverted to a station/split coverage path and the coverage path is a VDN, the switch will provide the following event reports for the call:

- Diverted Event Report - This event report is sent to a monitor request on a station. A Diverted Event Report can also be sent to the diverted-from VDN's monitor request on the call, if the diverted-to VDN in the coverage path has a monitor request. The diverted-to VDN's monitor request receives a Delivered (to an ACD device) Event Report. If the diverted-to VDN in the coverage path has no active monitor request (not monitored), then no Diverted Event Report is sent to the diverted-from VDN's monitor request for the call.
- Delivered (to ACD device) Event Report - This report is only sent if the diverted-to VDN in the call coverage path has a monitor request.

All other event reports associated with calls in a VDN (for example, Queued and Delivered Event Reports) are provided to all monitor requests on the call.

Call Forwarding All Calls

No Diverted Event Report will be sent to a cstaMonitorDevice request for the forwarding station, since the call does not alert the extension that has Call Forwarding activated. This is only if the call was placed directly to "forwarded to station."

If a monitored call is forwarded off-PBX over a non-PRI facility, the Network Reached Event Report will be generated.

Call Park

A cstaMonitorDevice-monitored station can activate Call Park.

A call may be parked manually at a station by use of the "call park" button (with or without the conference and/or transfer buttons), or by use of the feature access code and the conference and/or transfer buttons.

When a call is parked by using the "call park" button without either the conference or the transfer buttons, there are no event reports generated. When the conference or transfer buttons are used to park a call, the Conferenced or Transferred Event Reports are generated. In this case, the "calling" and the "called" number in the Conferenced or Transferred Event Reports will be the same as that of the station on which the call was parked.

When the call is unparked, an Established Event Report is generated with the "calling" and "called" numbers indicating the station on which the call had been parked, and the "connected" number is that of the station unparking the call.

Call Pickup

A call alerting at a cstaMonitorDevice-monitored station may be picked up using Call Pickup. The station picking up (either the principal or the pickup user or both) may be monitored. An Established Event Report is sent to all monitor requests on the call when this feature is used. When a pickup user picks up the principal's call, the principal's set (if multifunction) retains a simulated bridge appearance and is able to connect to the call at any time. No event report is sent for the principal unless the principal connects in the call.

When a call has been queued first and then picked up by a pickup user, it is possible for a client application to see an Established Event Report without having seen any prior Delivered Event Reports.

Call Vectoring

A VDN can have a monitor request. Interactions between event reporting and call vectoring are shown in Table 19.

Table 19: Interactions Between Feedback and Call Vectoring

Vector Step or Command	Event Report	When Sent	Cause
Vector Initialization	Delivered 88[1] (to ACD device)	encountered	
Queue to Main	Queued Failed	successfully queues queue full, no agents logged in	 queue full
Check Backup	Queued Failed	successfully queues queue full, no agents logged in	 queue full
Messaging Split	Queued Failed	successfully queues queue full, no agents logged in	 queue full
Announcement	none		
Wait	none		
GoTo	none		

Table 19: Interactions Between Feedback and Call Vectoring

Vector Step or Command	Event Report	When Sent	Cause
Stop	none		
Busy	Failed	Encountered	busy
Disconnect	Connection Cleared	Facility Dropped	busy
Go To Vector	none		
Route to (internal)	Delivered (to station device)		
Route To (external)	Network Reached		
Adjunct Routing	route		
Collected Digits	none		
Route To Digits (internal)	Delivered (to station device)		
Route To Digits (external)	Network Reached		
Converse Vector Command	Queued Event	If the call queues for the agent or automated attendant (VRU)	
	Delivered Event	When the call is delivered to an agent or the automated attendant	
	Established Event	When the call is answered by the agent or automated attendant	
	Connection Cleared Event	When the call disconnects from the agent or automated attendant	

1. Only reported over a VCN/ACD split monitor association.

Call Prompting

Up to 16 digits collected from the last "collect digit" vector command will be passed to the application in the Delivered Event Report. The collected digits are sent in the private data.

Lookahead Interflow

This feature is activated by encountering a ’’route to’’ vector command, with the route to destination being an off PBX number, and having the ISDN-PRI, Vectoring (Basic), and Lookahead Interflow options enabled on the Customer Options form.

For the originating PBX, the interactions are the same as with any call being routed to an off-PBX destination by the ’’route to’’ vector command.

For the receiving PBX, the lookahead interflow information element is passed in the ISDN message and will be included in all subsequent Delivered (to ACD device) Event Report for the call, when the information exists, and when the call is monitored. (Lookahead Interflow Information is supported in private data.)

Multiple Split Queueing

A Queued Event Report is sent for each split that the call queues to. Therefore, multiple call queued events could be sent to a client application for one call.

If a call is in multiple queues and abandons (caller drops), one Connection Cleared Event Report (cause normal) will be returned to the application followed by a Call Cleared Event Report.

When the call is answered at a split, the call will be removed from the other split’s queue. No other event reports for the queues will be provided in addition to the Delivered and Established Event Reports.

Call Waiting

When an analog station is administered with this feature and a call comes in while the user is busy on another call, the Delivered Event Report is sent to the client application.

Conference

Report is Manual conference from a cstaMonitorDevice monitored station is allowed, subject to the feature’s restrictions. The Held Event Report is provided as a result of the first button push or first switch-hook flash. The Conferenced Event Report is provided as a result of the second button push or second switch-hook flash, and only if the conference is successfully completed. On a manual conference or on a Conference Call Service request, the Conferenced Event is sent to all the monitor requests for the resultant call.

Consult

When the covering user presses the Conference or Transfer feature button and receives a dial tone, a Held Event Report is returned to monitor requests of the call. A Service Initiated Event Report is then returned to the monitor requests on the covering user. After the Consult button is pressed by the covering user, Delivered and Established Event Reports are returned to monitor requests on the principal and covering user. Then the covering user can conference or transfer the call.

CTI Link Failure

When the connectivity of the CTI link between the Communication Manager and the TSAPI Service is interrupted or reset, information of all calls received by the TSAPI Service before are not reliable. When CTI link failure happens, all call records are destroyed and information such as User To User Info, User Entered Code are deleted from the TSAPI Service. If the link is restored in time, the call events may resume for the new monitor requests (note that when CTI link is re-initialized, all monitor associations are aborted), but the Original Call Information for calls that exist before the link went down are not available.

Data Calls

Analog ports equipped with modems can be monitored by the cstaMonitorDevice Service and calls to and from ports can be monitored. However, Call Control Service requests may cause the call to be dropped by the modem.

DCS

With respect to event reporting, calls made over a DCS network are treated as off-PBX calls and only the Service Initiated, Network Reached, Call Cleared, and/or Connection Cleared Event Reports are generated. DCS/UDP extensions that are local to the PBX are treated as on-PBX stations. DCS/UDP extensions connected to the remote nodes are treated as off-PBX numbers.

Incoming DCS calls will provide a calling party number.

Direct Agent Calling and Number of Calls In Queue

Direct-agent calls will not be included in the calculation of number of calls queued for the Queued Event Report.

Drop Button Operation

The operation of this button is not changed with G3 CSTA Services.

When the ''Drop'' button is pushed by one party in a two-party call, the Connection Cleared Event Report is sent with the extension of the party that pushed the button. The originating party receives dial tone and the Service Initiated Event Report is reported on its cstaMonitorDevice requests.

When the ''Drop'' button is pushed by the controlling party in a conference, the Connection Cleared Event Report is sent with the extension of the party who was dropped off the call. This might be a station extension or a group extension. A group extension is provided in situations when the last added party to a conference was a group (for example, TEG, split, announcement, etc.) and the ''Drop'' button was used while the group extension was still alerting (or was busy). Since the controlling party does not receive dial tone (it is still connected to the conference), no Service Initiated Event Report is reported in this case.

Expert Agent Selection (EAS)

Logical Agents

Whenever logical agents are part of a monitored call, the following additional rules apply to the event reports:

- The callingDevice always contains the logical agent's physical station number (extension), even though a Make Call request might have contained a logical agent's login ID as the originating number (callingDevice).
- The answeringDevice and alertingDevice contain the logical agent's station extension and never contain the login ID. This is true regardless of whether the call was routed through a skill hunt group, whether the connected station has a logical agent currently logged in, or whether the call is an application-initiated or voice terminal-initiated direct agent call.
- The calledDevice contains the number that was dialed, regardless of the station connected to the call. For example, a call may be alerting an agent station, but the dialed number might have been a logical agent's login ID, a VDN, or another station.

- The Conferenced and Transferred Event Reports are an exception to this rule. In these events the addedParty contains the station extension of the transferred to or conferenced party when a local extension is involved. When an external extension is involved, the addedParty is unknown. If the transferred to or conferenced party is a hunt group or login ID and the call has not been delivered to a station, the addedParty contains the hunt group or login ID extension. If the call has been delivered to a station, the addedParty contains the station extension connected to the call.
- The alertingDevice in the Delivered and the queue in the Queued Event Report for logical direct agent calls contains a skill hunt group from the set of skills associated with the logical agent. Note that the skill hunt group is provided, even though an application-initiated, logical direct agent call request did not contain a skill hunt group.

Hold

Manually holding a call (either by using the Hold, Conference, Transfer buttons, or switch-hook flash) results in the Held Event Report being sent to all monitor requests for this call, including the held device. A held party is considered on the call for the purpose of receiving events relevant to that call.

Integrated Services Digital Network (ISDN)

The Make Call calls will follow Integrated Services Digital Network (ISDN) rules for the originator's name and number. The Service Initiated Event Report will not be sent for en-bloc BRI sets.

Multiple Split Queueing

When a call is queued in multiple ACD splits and then removed from the queue, the Delivered Event Report will provide the split extension of the alerting agent. There will be no other events provided for the splits from which the call was removed.

Personal Central Office Line (PCOL)

Members of a Personal Central Office Line (PCOL) may be monitored by the cstaMonitorDevice Service. PCOL behaves like bridging for the purpose of event reporting. When a call is placed to a PCOL group, the Delivered Event Report is provided to each member's cstaMonitorDevice requests. The calledDevice information passed in the Delivered event will be the default station characters. When one of the members answers the incoming call, the Established Event Report provides the extension of the station that answered the call. If another member connects to the

call, another Established Event Report is provided. When a member goes on hook but the PCOL itself does not drop from the call, no event is sent but the state of that party changes from the connected state to the bridged state. The Connection Cleared Event Report is not sent to each member's cstaMonitorDevice requests until the entire PCOL drops from the call (as opposed to an individual member going on-hook). Members that are not connected to the call while the call is connected to another PCOL member are in the bridged state. When the only connected member of the PCOL transitions to the held state, the state for all members of the PCOL changes to the held state even if they were previously in bridged state. There is no event report sent to any cstaMonitorDevice request(s) for bridged users for this transition.

All members of the PCOL may be individually monitored by the cstaMonitorDevice Service. Each will receive appropriate events as applicable.

Primary Rate Interface (PRI)

Primary Rate Interface (PRI) facilities may be used for either inbound or outbound application monitored calls.

An incoming call over a PRI facility will provide the callingDevice and calledDevice information (CPN/BN/DNIS) which is passed on to the application in the Delivered (to ACD device) and Established Event Reports.

An outgoing call over a PRI facility provides call feedback events from the network.

A cstaMakePredictiveCall call will always use a call classifier on PRI facilities, whether the call is interworked or not. Although these facilities are expected to report call outcomes on the ''D'' channel, often interworking causes loss or delay of such reports. Progress messages reporting ''busy,'' SITs, ''alert,'' and ''drop/disconnect'' will cause the corresponding event report to be sent to the application. For cstaMakePredictiveCall calls, the ''connected'' number is interpreted as ''far end answer'' and is reported to the application as the Established Event Report when received before the call classifiers' ''answer'' indication. When received after the call classifier has reported an outcome, it will not be acted upon. A monitored outbound call over PRI facilities may generate the Delivered, Established, Connection Cleared, and/or Call Cleared Event Reports, if such a call goes ISDN end-to-end. If such a call interworks, the ISDN PROGress message is mapped into a Network Reached Event Report. In this case, only the Connection Cleared or Call Cleared Event Reports may follow.

Ringback Queueing

CstaMakePredictiveCall calls will be allowed to queue on busy trunks or stations.

When activated, the callback call will report events on the same callID as the original call.

Send All Calls (SAC)

For incoming calls, the Delivered Event Report is sent only for multifunction sets receiving calls while having SAC activated. The Delivered Event Report is not generated for analog sets when the SAC feature is activated and the set is receiving a call.

Service-Observing

CstaMonitorDevice monitored stations may be service-observed and observers. When a monitored station is the observer, and it is bridged onto a call for the purpose of service observing, the Established Event Report is sent to the observer's cstaMonitorDevice requests for as well as to all other monitor requests for that call.

Temporary Bridged Appearances

The operation of this feature has not changed with G3 CSTA Services. There is no event provided when a temporary bridged appearance is created at a multifunction set. If the user is connected to the call (becomes active on such an appearance), the Established Event Report is provided. If a user goes on hook after having been connected on such an appearance, a Connection Cleared Event Report (normal clearing) is generated for the disconnected extension (bridged appearance).

If the call is dropped from the temporary bridged appearance by someone else, a Connection Cleared Event Report is also provided.

Temporary bridged appearances are not supported with analog sets. Analog sets get the Diverted Event Report when such an appearance would normally be created for a multifunction set.

The call state provided to queries about extensions with temporary bridged appearances will be ''bridged'' if the extension is not active on the call or it will be ''connected'' if the extension is active on the call.

Terminating Extension Group (TEG)

Members of a TEG may be monitored by the cstaMonitorDevice Service. A TEG behaves similarly to bridging for the purpose of event reporting. If cstaMonitorDevice monitored stations are members of a terminating group, an incoming call to the group will cause a Delivered Event Report to be sent to all cstaMonitorDevice requests for members of the terminating group. On the cstaMonitorDevice request for the member of the group that answers the call, an Established Event Report is returned to the answering member's cstaMonitorDevice request(s)

which contains the station that answered the call. All the cstaMonitorDevice requests for the other group members (nonanswering members without TEG buttons) receive a Diverted Event Report. When a button TEG member goes on hook but the TEG itself does not drop from the call, no event is sent but the state of that party changes from the connected state to the bridged state.

The Connection Cleared Event Report is not sent to each member's cstaMonitorDevice requests until the entire TEG drops from the call (as opposed to an individual member going on hook).

Members that are not connected to the call while the call is connected to another TEG member are in the bridged state. When the only connected member of the TEG transitions to the held state, the state for all members of the TEG changes to the held state even if they were previously in the bridged state. There is no event report sent over the cstaMonitorDevice requests for the bridged user(s) for this transition.

All members of the TEG may have individual cstaMonitorDevice request. Each will receive appropriate events as applicable to the monitored station.

Transfer

Manual transfer from a station monitored by a cstaMonitorDevice request is allowed subject to the feature's restrictions. The Held Event Report is provided as a result of the first button push (or switch-hook flash for analog sets). The Transferred Event Report is provided as a result of the second button push (or on-hook for analog sets), and only if the transfer is successfully completed. The Transferred Event Report is sent to all monitor requests for the resultant call.

Trunk-to-Trunk Transfer

Existing rules for trunk-to-trunk transfer from a station user will remain unchanged for monitored calls. In such cases, transfers requested via Transfer Call request will be negatively acknowledged. When this feature is enabled, monitored calls transferred from trunk-to-trunk will be allowed, but there will be no further notification.

Chapter 12: Routing Service Group

Routing Service Group describes the services that allow the switch to request and receive routing instructions for a call. These instructions, issued by a client routing server application, are based upon the incoming call information provided by the switch. The following Routing Services are available:

Route End Event

Summary

- Direction: Switch to Client
- Event: CSTARouteEndEvent
- Service Parameters: routeRegisterReqID, routingCrossRefID, errorValue

Functional Description:

This event is sent by the switch to terminate a routing dialog for a call and to inform the routing server application of the outcome of the call routing.

Service Parameters:

routeRegisterReqID [mandatory] Contains the handle to the routing registration session for which the application is providing routing services. The application received this handle in a CSTARouteRegisterReqConfEvent confirmation to a cstaRouteRegisterReq() request.

routingCrossRefID [mandatory] Contains the handle to the CSTA call routing dialog for a call. The application previously received this handle in the CSTARouteRequestExtEvent for the call. This is the routing dialog that the switch is ending.

errorValue [mandatory] Contains the cause code for the reason why the switch is ending the routing dialog. One of the following values will be returned:

- GENERIC_UNSPECIFIED (0) (CS0/16)
  - The call has been routed successfully.
  - The adjunct route request to route using NCR resulted in the call not being routed by NCR because of an internal system error.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (CS0/50) The adjunct route request to route using NCR resulted in the call not being routed by NCR because the NCR contained incorrectly administered trunk (NCR is active but not set up correctly).
- INVALID_CALLING_DEVICE (5) (CS3/15) Upon routing to an agent (for a direct-agent call), the agent is not logged in.
- PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE (8) (CS3/43) Lack of calling permission; for example, for an ARS call, there is an insufficient Facility Restriction Level (FRL). For a direct-agent call, the originator's Class Of Restriction (COR) or the destination agent's COR does not allow a direct-agent call.

- INVALID_DESTINATION (14) (CS0/28) The destination address in the cstaRouteSelectInv() is invalid.
- The adjunct route request to route using NCR resulted in the call not being routed by NCR because the NCR contained in invalid PSTN number
- INVALID_OBJECT_TYPE (18) (CS3/11) Upon routing to an agent (for direct-agent call), the agent is not a member of the specified split.
- INVALID_OBJECT_STATE (22) A Route Select request was received by the TSAPI Service in wrong state. A second Route Select request sent by the application before the routing dialog is ended may cause this.
- NETWORK_BUSY (35) (CS0/34) The adjunct route request to route using NCR resulted in the call not being routed by NCR because there was no NCT outgoing trunk.
- NETWORK_OUT_OF_SERVICE (36) (CS3/38)
  - The adjunct route request to route using NCR resulted in the call not being routed by NCR because the NCT contained an invalid PSTN number, and the second leg can not be set up.
  - The adjunct route request to route using NCR resulted in the call not being routed by NCR because of a PSTN NCD network error.
  - .The adjunct route request to route using NCR resulted in the call not being routed by NCR because of a PSTN NCD no disc error.
- NO_ACTIVE_CALL (24) (CS0/86, CS3/86) The call was dropped (for example, caller abandons, vector disconnect timer times out, a non-queued call encounters a "stop" step, or the application clears the call) while waiting for a cstaRouteSelectInv() response.
- NO_CALL_TO_ANSWER (28) (CS3/30) The call has been redirected. The switch has canceled or terminated any outstanding CSTARouteRequestExtEvent (s) for the call after receiving the first valid cstaRouteSelectInv() message. The switch sends a Route End Event with this cause to all other outstanding CSTARouteRequestExtEvent (s) for the call. Note that this error can happen when Route Registers are registered for the same routing device from two different AE Servers and the switch is set to send multiple Route Requests for the same call.
- PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE (8) (CS3/43) The adjunct route request to route using NCR resulted in the call not being routed by NCR because the PSTN NCD exceeds the maximum redirections.

- RESOURCE_BUSY (33) (CS0/17) The destination is busy and does not have coverage. The caller will hear either a reorder or busy tone.
- PERFORMANCE_LIMIT_EXCEEDED (52) (CS0/102) Call vector processing encounters any steps other than wait, announcement, goto, or stop after the CSTARouteRequextExtEvent (adjunct routing command) has been issued. This can also happen when a wait step times out. When the switch sends CSTARouteEndEvent with this cause, call vector processing continues.
- VALUE_OUT_OF_RANGE (3) (CS0/96) The adjunct route request to route using NCR resulted in the call not being routed by NCR because Route Select does not contain a called number.

Detailed Information:

An application may receive one Route End Event and one Universal Failure for a Route Select request for the same call in one of the following call scenarios:

- Switch/TSAPI Service sends a Route Request to application.
- Caller drops the call.
- Application sends a Route Select Request to TSAPI Service.
- Switch/TSAPI Service sends a Route End Event (errorValue = NO_ACTIVE_CALL) to application.
- TSAPI Service receives the Route Select Request, but call has been dropped.
- TSAPI Service sends Universal Failure for the Route Select request (errorValue = INVALID_CROSS_REF_ID) to application.

Syntax

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

// CSTARouteEndEvent - Route Select Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTAEVENTREPORT
            EventType_t eventType;  // CSTA_ROUTE_END
} ACSEventHeader_t;

typedef struct  CSTARouteEndEvent_t {
            RouteRegisterReqID_t    routeRegisterReqID,
            RoutingCrossRefID_t     routingCrossRefID,
            CSTAUniversalFailure_t  errorValue,
} CSTARouteEndEvent_t;

typedef struct
{
            ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
                                    union
                                    {
                                    CSTARouteEndEvent_t  routeEnd;
                                    } u;
                        } cstaEventReport;
            } event;
char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Route End Service (TSAPI Version 2)

Summary

- Direction: Client to Switch
- Function: cstaRouteEndInv()
- Service Parameters: routeRegisterReqID, routingCrossRefID, errorValue
- Ack Parameters: noData
- Nak Parameter: universalFailure

Functional Description:

This service is sent by the routing server application to terminate a routing dialog for a call. The service request includes a cause value giving the reason for the routing dialog termination.

Service Parameters:

routeRegisterReqID	[mandatory] Contains the handle to the routing registration session for which the application is providing routing services. The routing server application received this handle in a CSTARouteRegisterReqConfEvent confirmation to a cstaRouteRegisterReq() request.
routingCrossRefID	[mandatory] Contains the handle to the CSTA call routing dialog for a call. The routing server application previously received this handle in the CSTARouteRequestExtEvent for the call. This is the routing dialog that the application is terminating.
errorValue	[mandatory] Contains the cause code for the reason why the application is terminating the routing dialog. One of the following values can be sent: • Any CSTA universalFailure error code • The errorValue is ignored by Communication Manager and has no effect for the routed call, but it must be present in the API. Suggested error codes that may be useful for error logging purposes are: • GENERIC_UNSPECIFIED (0) Normal termination (for example, application does not want to route the call or does not know how to route the call). • INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid routeRegisterReqID has been specified in the RouteEndInv() request. • RESOURCE_BUSY (33) Routing server is too busy to handle the route request. • RESOURCE_OUT_OF_SERVICE (34) Routing service temporarily unavailable due to internal problem (for example, the database is out of service).

Ack Parameters:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786. INVALID_CROSS_REF_ID (17) An invalid routeRegisterReqID or routeCrossRefID has been specified in the Route Ended request.

Detailed Information:

- If an application terminates a Route Request via a cstaRouteEndInv(), the switch continues vector processing.
- An application may receive one Route End Event and one Universal Failure for a cstaRouteEndInv() request for the same call in the following call scenario:
- Switch/TSAPI Service sends a CSTARouteRequestEvent to application.
- Caller drops the call.
- Application sends a cstaRouteEndInv() request to TSAPI Service.
- Switch/TSAPI Service sends a CSTARouteEndEvent (errorValue = NO_ACTIVE_CALL) to application.
- TSAPI Service receives the cstaRouteEndInv() request, but call has been dropped.
- TSAPI Service sends universalFailure for the cstaRouteEndInv() request (errorValue = INVALID_CROSS_REF_ID) to application.

Syntax

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

// cstaRouteEndInv() - Service Request

RetCode_t   cstaRouteEndInv (
            ACSHandle_t             acsHandle,
            InvokeID_t              invokeID,
            RouteRegisterReqID_t    routeRegisterReqID,
            RoutingCrossRefID_t     routingCrossRefID,
            CSTAUniversalFailure_t  errorValue,
            PrivateData_t           *privateData);

typedef long            RouteRegisterReqID_t;

typedef long            RoutingCrossRefID_t;
```

Route End Service (TSAPI Version 1)

Summary

- Direction: Client to Switch
- Function: cstaRouteEnd()
- Service Parameters: routeRegisterReqID, routingCrossRefID, errorValue

Functional Description:

This service is sent by the routing server application to terminate a routing dialog for a call. The service request includes a cause value giving the reason for the routing dialog termination.

Detailed Information:

An application may receive two CSTARouteEndEvent(s) for the same call in one of the following call scenarios:

- Switch/TSAPI Service sends a CSTARouteRequestEvent to application.
- Caller drops the call.
- Application sends a cstaRouteSelect() to TSAPI Service.
- Switch/TSAPI Service sends a CSTARouteEndEvent (errorValue = NO_ACTIVE_CALL) to application.
- TSAPI Service receives the cstaRouteSelect() Request, but call has been dropped.
- TSAPI Service sends CSTARouteEndEvent (errorValue = INVALID_CROSS_REF_ID) to application.

Syntax

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

// cstaRouteEnd() - Service Request

RetCode_t   cstaRouteEnd (
            ACSHandle_t             acsHandle,
            RouteRegisterReqID_t    routeRegisterReqID,
            RoutingCrossRefID_t     routingCrossRefID,
            CSTAUniversalFailure_t  errorValue,
            PrivateData_t           *privateData);

typedef long            RouteRegisterReqID_t;

typedef long            RoutingCrossRefID_t;
```

Route Register Abort Event

Summary

- Direction: Switch to Client
- Event: CSTARouteRegisterAbortEvent
- Service Parameters: routeRegisterReqID

Functional Description:

This event notifies the application that the TSAPI Service or switch aborted a routing registration session. After the abort occurs, the application receives no more CSTARouteRequestExtEvent(s) from this routing registration session and the routeRegisterReqID is no longer valid. The routing requests coming from the routing device will be sent to the default routing server, if a default routing registration is still active.

Service Parameters:

routeRegisterReqID	[mandatory] Contains the handle to the routing registration session for which the application is providing routing services. The application received this handle in a CSTARouteRegisterReqConfEvent confirmation to a cstaRouteRegisterReq() request.

Detailed Information:

- If no CTI link has ever received any CSTARouteRequestExtEvent(s) for the registered routing device and all of the CTI links are down, this event is not sent.
- In a multi-link configuration, if at least one link that has received at least one CSTARouteRequestExtEvent for the registered routing device is up, this event is not sent. It is sent only when all of the CTI links that have received at least one CSTARouteRequestExtEvent for the registered routing device are down.

  Note:
  How Communication Manager sends the CSTARouteRequestExtEvent(s) for the registered routing device, via which CTI links, is controlled by the call vectoring administered on the switch. A routing device can receive CSTARouteRequestExtEvent(s) from different CTI links. It is possible that links are up and down without generating this event.
- If the application wants to continue the routing service after the CTI link is up, it must issue a cstaRouteRegisterReq() to re-establish a routing registration session for the routing device.
- The Route Register Abort Event is sent when a competing application sends a route request and it has the same criteria (login, application name, and IP address).

Syntax

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

// CSTARouteRegisterAbortEvent

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTAEVENTREPORT
            EventType_t eventType;  // CSTA_ROUTE_REGISTER_ABORT
} ACSEventHeader_t;

typedef struct  CSTARouteRegisterAbortEvent_t {
            ACSEventHeader_t        eventHeader;
            union
            {           struct
                        {
                                    union
                                    {
                        CSTARouteRegisterAbortEvent_trouteCancel;
                                    } u;
                        } cstaEventReport;
            } event;
} CSTAEvent_t;

typedef  struct  CSTARouteRegisterAbortEvent_t {
            RouteRegisterReqID_t    routeRegisterReqID,
} CSTARouteRegisterAbortEvent_t;

typedef     long        RouteRegisterReqID_t;
```

Route Register Cancel Service

Summary

- Direction: Client to Switch
- Function: cstaRouteRegisterCancel()
- Confirmation Event: CSTARouteRegisterCancelConfEvent
- Service Parameters: routeRegisterReqID
- Ack Parameters: noData
- Nak Parameter: universalFailure

Functional Description:

Client applications use cstaRouteRegisterCancel() to cancel a previously registered cstaRouteRegisterReq() session. When this service request is positively acknowledged, the client application is no longer a routing server for the specific routing device and the TSAPI Service stops sending CSTARoutingRequestEvent(s) from the specific routing device associated with the routeRegisterReqID to the requesting client application. The TSAPI Service will send any further CSTARoutingRequestEvent(s) from the routing device to the default routing server application, if there is one registered.

Service Parameters:

routeRegisterReqID	[mandatory] Contains the handle to the routing registration session for which the application is canceling. The routing server application received this handle in a CSTARouteRegisterReqConfEvent confirmation to a cstaRouteRegisterReq() request.

Ack Parameters:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786. • INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid routeRegisterReqID has been specified in the request.

Detailed Information:

An application may receive CSTARouteRequestExtEvent after a cstaRouteRegisterCancel request is sent and before a CSTARouteRegisterCancelConfEvent response is received. The application should ignore the CSTARouteRequestExtEvent. If a cstaRouteSelectInv() request is sent for the CSTARouteRequestExtEvent, a CSTARouteEndEvent response will be received with error INVALID_CSTA_DEVICE_IDENTIFIER. If a cstaRouteEndInv() request is sent for the CSTARouteRequestExtEvent, it will be ignored. The outstanding CSTARouteRequestExtEvent will receive no response and will be timed out eventually.

Syntax

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

//  cstaRouteRegisterCancel() - Service Request

RetCode_t   cstaRouteRegisterCancel (
            ACSHandle_t             acsHandle,
            InvokeID_t              invokeID,
            RouteRegisterReqID_t    routeRegisterReqID,
            PrivateData_t           *privateData);

typedef long            RouteRegisterReqID_t;


//  CSTARouteRegisterCancelConfEvent - Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTACONFIRMATION
            EventType_t eventType;  // CSTA_ROUTE_REGISTER_CANCEL_CONF
} ACSEventHeader_t;

typedef struct CSTARouteRegisterCancelConfEvent_t {
            RouteRegisterReqID_t routeRegisterReqID;
} CSTARouteRegisterCancelConfEvent_t;

typedef struct
{
            ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
                                    InvokeID_t  invokeID;
                                    union
                                    {
            CSTARouteRegisterCancelConfEvent_trouteCancel;
                                    } u;
                        } cstaConfirmation;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Route Register Service

Summary

- Direction: Client to Switch
- Function: cstaRouteRegisterReq()
- Service Parameters: routingDevice
- Ack Parameters: routeRegisterReqID
- Nak Parameter: universalFailure

Functional Description:

Client applications use cstaRouteRegisterReq() to register as a routing server for CSTARouteRequestExtEvent from a specific device. The application must register for routing services before it can receive any CSTARouteRequestExtEvent(s) from the routing device. An application may be a routing server for more that one routing device. For a specific routing device,however, the TSAPI Service allows only one application registered as the routing server.

If a routing device already has a routing server registered, subsequent cstaRouteRegisterReq() requests will be negatively acknowledged, except as described in Special usage cases. This special usage is introduced with AE Services 4.0.

Special usage cases: In some cases it is desirable to allow the same application to re-register as a routing device. That is, if a routing device already has a routing server registered, subsequent cstaRouteRegisterReq() requests will be positvely acknowledged if certain criteria conditions are satisfied. For example, if a link goes down with an AE Services application, the application can re-establish itself if the following criteria are met:

- If the login (LoginID_t) matches that of the previously registered application
- If the application name (AppName_t) matches that of the previously registered application
- If the IP address of the client machine matches matches that of the previously registered application

Service Parameters:

routingDevice	[mandatory] Contains the device identifier of the routing device for which the application requests to be the routing server. A valid routing device on a G3 switch is a VDN extension which has the proper routing vector step set up to send the Route Requests to a TSAPI Service. A NULL device identifier indicates that the requesting application will be the default routing server for Communication Manager. A default routing server will receive CSTARouteRequestExtEvent(s) from routing devices of Communication Manager that do not have a registered routing server.

Ack Parameters:

routeRegisterReqID	[mandatory] Contains a handle to the routing registration session for a specific routing device (or for the default routing server). All routing dialogs (identified by routingCrossRefID [s]) for a routing device occur over this routing registration session.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786. OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44) The specified routing device already has a registered routing server.

Detailed Information:

- The cstaRouteRegisterReq() is handled by the TSAPI Service, not by Communication Manager. The Route Requests are sent from the switch to the TSAPI Service through call vector processing. From the perspective of the switch, the TSAPI Service is the routing server. The TSAPI Service processes the Route Requests and sends the CSTARouteRequestExtEvent(s) to the proper routing servers based on the route registrations from applications.
- If no routing server is registered for Communication Manager, all Route Requests from the switch will be terminated by the TSAPI Service with a Route End Request, as if cstaRouteEndInv() requests were received from a routing server.

Syntax

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

//  cstaRouteRegisterReq() - Service Request

RetCode_t   cstaRouteRegisterReq (
            ACSHandle_t             acsHandle,
            InvokeID_t              invokeID,
            DeviceID_t              *routingDevice,
            PrivateData_t           *privateData);

typedef long           RouteRegisterReqID_t;

//  CSTARouteRegisterReqConfEvent - Service Response

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

typedef struct CSTARouteRegisterReqConfEvent_t {
            RouteRegisterReqID_t    registerReqID;
} CSTARouteRegisterReqConfEvent_t;

typedef struct
{
   ACSEventHeader_teventHeader;
   union
   {
            struct
            {
                        InvokeID_t  invokeID;
                        union
                        {
            CSTARouteRegisterReqConfEvent_trouteRegister;
                        } u;
            } cstaConfirmation;
   } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Route Request Service (TSAPI Version 2)

Summary

- Direction: Switch to Client
- Event: CSTARouteRequestExtEvent
- Private Data Event: ATTRouteRequestEvent, ATTV6RouteRequestEvent (private data version 6), ATTV5RouteRequestEvent (private data version 5), ATTV4RouteRequestEvent (private data versions 2, 3, and 4)
- Service Parameters: *routeRegisterReqID, routingCrossRefID, currentRoute, callingDevice, routedCall, routedSelAlgorithm, priority, setupInformation*
- Private Parameters: *trunkGroup, trunkMember, lookaheadInfo, userEnteredCode, userInfo, ucid, callOrigintorInfo, flexibleBilling, deviceHistory*
- Ack Parameters: N/A; see cstaRouteSelectInv() (Route Select Service (TSAPI Version 2))
- Nak Parameter: N/A; see cstaRouteEndInv() (Route End Service (TSAPI Version 2))

Functional Description:

The switch sends a CSTARouteRequestExtEvent to request a destination for a call arrived on a routing device from a routing server application. The application may have registered as the routing server for the routing device on the switch that is making the request, or it may have registered as the default routing server. The CSTARouteRequestExtEvent includes call-related information. A routing server application typically uses the call-related information and a database to determine the destination for the call. The routing server application responds to the CSTARouteRequestExtEvent via a cstaRouteSelectInv() request that specifies a destination for the call or a cstaRouteEndInv() request, if the application has no destination for the call.

Service Parameters:

routeRegisterReqID	[mandatory] Contains a handle to the routing registration session for which the application is providing routing service. The routing server application received this handle in a CSTARouteRegisterReqConfEvent confirmation to a cstaRouteRegisterReq() request.
routingCrossRefID	[mandatory] Contains the handle for the routing dialog of this call. This identifier is unique within a routing session identified by the routeRegisterReqID.
currentRoute	[mandatory] Specifies the destination of the call. This is the VDN extension number first entered by the call (see Detailed Information:).
callingDevice	[optional - supported] Specify the call origination device. This is the calling device number for on-PBX originators or incoming calls over PRI facilities. For incoming calls over non-PRI facilities, the trunk identifier is provided. **Note:** The trunk identifier is a dynamic device identifier. It cannot be used to access a trunk in Communication Manager
routedCall	[optional - supported] Specifies the callID of the call that is to be routed. This is the connectionID of the routed call at the routing device.
routedSetAlgorithm	[optional - partially supported] Indicates the type of routing algorithm requested. Type is set to SV_NORMAL.
priority	[optional - not supported] Indicates the priority of the call and may affect selection of alternative routes.
setupInformation	[optional - not supported] Contains an ISDN call setup message if available.

Private Parameters:

trunkGroup	[optional] Specifies the trunk group number from which the call is originated. The callingDevice and trunk parameters are mutually exclusive. Beginning with G3V8, both the calling device and trunk group may be present. Prior to G3V8, one or the other will be present, but not both. This parameter is supported by private data version 5 and later only.
trunkMember	[optional] This parameter is supported beginning with G3V4. It specifies the trunk member number from which this call originated. Beginning with G3V8, trunk member number is provided regardless of whether the calling device is available. Prior to G3V8, trunkMember number is provided only if the calling device is unavailable
trunk	[optional] Specifies the trunk group number from which the call is originated. The callingDevice and trunk parameters are mutually exclusive. One or the other will be present, but not both. This parameter is supported by private data versions 2, 3, and 4.
lookaheadInfo	[optional] Specifies the lookahead interflow information received from the incoming call that is to be routed. The lookahead interflow is a G3 switch feature that routes some of the incoming calls from one switch to another so that they can be handled more efficiently and will not be lost. The switch that overflows the call provides the lookahead interflow information. The routing server application may use the lookahead interflow information to determine the destination of the call. Please refer to the DEFINITY Generic 3 Feature Description for more information about lookahead interflow. If the lookahead interflow type is set to "LAI_NO_INTERFLOW", no lookahead interflow private data is provided with this event.
userEnteredCode	[optional] Specifies the code/digits that may have been entered by the caller through the G3 call prompting feature or the collected digits feature. If the userEnteredCode code is set to "UE_NONE", no userEnteredCode private data is provided with this event.
userInfo	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string. Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes. **Note:** An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch. The following UUI protocol types are supported: ● UUI_NONE — There is no data provided in the data parameter. ● UUI_USER_SPECIFIC — The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter. ● UUI_IA5_ASCII — The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

ucid	[optional] Specifies the Universal Call ID (UCID) of the routed call. The UCID is a unique call identifier across switches and the network. A valid UCID is a null-terminated ASCII character string. If there is no UCID associated with this call, the ucid contains the ATT_NULL_UCID (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.
callOriginator	[optional] Specifies the callOriginatorInfo of the call originator such as coin call, 800-service call, or cellular call. This information is from the network, not from the DEFINITY switch. The type is defined in Bellcore publication "Local Exchange Routing Guide" (document number TR-EOP-000085). A list of the currently defined codes, as of June 1994, is in the Detailed Information sub-section of the "Delivered Event" section. This parameter is supported by private data version 5 and later only.
flexibleBilling	[optional] Specifies whether the Flexible Billing feature is allowed for this call and the Flexible Billing customer option is assigned on the switch. If this parameter is set to TRUE, the billing rate can be changed for the incoming 900-type call using the Set Bill Rate Service. This parameter is supported by private data version 5 and later only.
DeviceHistory	The DeviceHistory parameter type specifies a list of deviceIDs that were previously associated with the call. A device becomes associated with the call whenever there is a CSTA connection created at the device for the call. The association may also result from a relationship between a device and a call outside the CSTA switching function. A device becomes part of the DeviceHistory list when it is no longer associated with the call (for example: when a call is redirected from a device, when a call is transferred away from a device, and when a device clears from a call).

The DeviceHistory parameter consists of a list of entries. Each entry contains information about a deviceID that had previously been associated with the call. The list is ordered from the first device that left the call to the device that most recently left the call.

- **oldDeviceID (M) DeviceID** - the device that left the call. This information should be consistent with the subject device in the event that represented the device leaving the call. For example: the divertingDevice provided in the Diverted event for that redirection, the transferring device in the Transferred event for a transfer, or the clearing device in the Connection Cleared event. This device identifier type may be one of the following:
  - of any device identifier format.
  - "Not Known" - indicates that the device identifier associated with this entry in the DeviceHistory list cannot be provided.
  - "Restricted" - indicates that the device associated with this entry in the DeviceHistory list cannot be provided due to regulatory and/or privacy reasons.
  - "Not Required" - indicates that there are no devices that have left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID are not provided with this list entry.
  - "Not Specified" - indicates that the switching function cannot determine whether or not any devices have previously left the call. If this value is provided, it is provided as the only entry in the list and the eventCause and oldConnectionID is not be provided with this list entry.
- **EventCause (O) EventCause** - the reason the device left the call or was redirected. This information should be consistent with the eventCause provided in the event that represented the device leaving the call (for example, the cause code provided in the Diverted, Transferred, or Connection Cleared event).
- **OldConnectionID (O) ConnectionID** - the CSTA connectionID that represents the last connectionID associated with the device that left the call. This information should be consistent with the subject connection in the event that represented the device leaving the call (for example, the connectionID provided in the Diverted, Transferred, or Connection Cleared event).

  Note: Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be EC_NETWORKSIGNAL if a ISDN Redirected Number was provided; otherwise the cause value is set to match the cause value of the event that was flowed to report the dropped connection.

Ack Parameters:

N/A	See cstaRouteSelectInv().

Nak Parameter:

N/A	See cstaRouteEndInv().

Detailed Information:

- The Routing Request Service can only be administered through the Basic Call Vectoring feature. The switch initiates the Routing Request when the Call Vectoring processing encounters the adjunct routing command in a call vector. The vector command will specify a CTI link's extension through which the switch will send the Route Request to the TSAPI Service.
- Multiple adjunct routing commands are allowed in a call vector. In G3V3, the Multiple Outstanding Route Requests feature allows 16 outstanding Route Requests per call. The Route Requests can be over the same or different CTI links. The requests are all made from the same vector. They may be specified back-to-back, without intermediate (wait, announcement, goto, or stop) steps. If the adjunct routing commands are not specified back-to-back, pre-G3V3 adjunct routing functionality will apply. This means that previous outstanding Route Requests are canceled when an adjunct routing vector step is executed.
- The first Route Select response received by the switch is used as the route for the call, and all other Route Requests for the call are canceled via CSTARouteEndEvent(s).
- If an application terminates the CSTARouteRequestExtEvent request via a cstaRouteEndInv(), the switch continues vector processing.
- A CSTARouteRequestExtEvent request will not affect the Call Event Reports.
- Like Delivered or Established Event, the Route Request currentRoute parameter contains the called device. In release 1 and release 2, the currentRoute in Route Request contains the originally called device if there is no distributing device, or the distributing device if the call vectoring with VDN override feature of the PBX is turned on. In the later case, the originally called device is not reported. The distributingDevice feature is not supported in the Route Request private data. See the "Delivered Event" section for detailed information on the distributingDevice parameter.

Syntax

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

//  CSTARouteRequestExtEvent - CSTA Unsolicited Event

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTAREQUEST
            EventType_t eventType;  // CSTA_ROUTE_REQUEST_EXT
} ACSEventHeader_t;

typedef long            RouteRegisterReqID_t;

typedef long            RoutingCrossRefID_t;

typedef char            DeviceID_t[64];

typedef struct ExtendedDeviceID_t {
            DeviceID_t      deviceID;
            DeviceIDType_t  deviceIDType;
            DeviceIDStatus_tdeviceIDStatus;
} ExtendedDeviceID_t;

typedef ExtendedDeviceID_t CallingDeviceID_t;

typedef ExtendedDeviceID_t CalledDeviceID_t;

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

typedef struct ConnectionID_t {
            long                    callID;
            DeviceID_t              deviceID;
            ConnectionID_Device_t   devIDType;
} ConnectionID_t;

typedef enum SelectValue_t {
            SV_NORMAL = 0,
            SV_LEAST_COST = 1,
            SV_EMERGENCY = 2,
            SV_ACD = 3,
            SV_USER_DEFINED = 4
} SelectValue_t;
```

Syntax (Continued)

```
typedef struct SetupValues_t {
            int                     length;
            unsigned char           *value;
} SetupValues_t;

typedef struct CSTARouteRequestExtEvent_t  {
            RouteRegisterReqID_t    routeRegisterReqID;
            RoutingCrossRefID_t     routingCrossRefID;

            CalledDeviceID_tcurrentRoute;// TSAPI V1 and V2 are
                                                 // different
            CallingDeviceID_tcallingDevice;// TSAPI V1 and V2 are
                                                 // different
            ConnectionID_t          routedCall;
            SelectValue_t           routedSelAlgorithm;
            Boolean                 priority;
            SetUpValues_t           setupInformation;
} CSTARouteRequestExtEvent_t;

typedef struct
{
            ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
            InvokeID_t  invokeID;// Unused for Route Request Event
                                    union
                                    {
                        CSTARouteRequestExtEvent_trouteRequestExt;
                                    } u;
                        } cstaRequest;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Data Version 7 and 8 Syntax

If private data accompanies a CSTARouteRequestExtEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTARouteRequestExtEvent does not deliver private data to the application.

If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this Route Request Event.

The DeviceHistory parameter is added for private data version 7.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTVRouteRequestEvent - CSTA Unsolicited Event Private Data

typedef struct ATTRouteRequestEvent_t {
    DeviceID_t      trunkGroup;
    ATTLookaheadInfo_t lookaheadInfo;
    ATTUserEnteredCode_t userEnteredCode;
    ATTUserToUserInfo_t userInfo;
    ATTUCID_t       ucid;
    ATTCallOriginatorInfo_t callOriginatorInfo;
    unsigned char   flexibleBilling;
    DeviceID_t      trunkMember;
    DeviceHistory_t deviceHistory;
} ATTRouteRequestEvent_t;
```

Private Data Version 6 Syntax

If private data accompanies a CSTARouteRequestExtEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTARouteRequestExtEvent does not deliver private data to the application.

If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this Route Request Event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV6RouteRequestEvent - CSTA Unsolicited Event Private Data

typedef struct
{
            ATTEventTypeeventType;  // ATTV6_ROUTE_REQUEST
            union
            {
                        ATTV6RouteRequestEvent_t v6routeRequest;
            } u;
} ATTEvent_t;

typedef struct ATTRouteRequestEvent_t
{
            DeviceID_t              trunkGroup;
            ATTLookaheadInfo_t      lookaheadInfo;
            ATTUserEnteredCode_t    userEnteredCode;
            ATTUserToUserInfo_t     userInfo;
            ATTUCID_t               ucid;
            ATTCallOriginatorInfo_t callOriginatorInfo;
            Boolean                 flexibleBilling;
            DeviceID_t              trunkMember;
} ATTRouteRequestEvent_t;

typedef char ATTUCID_t[64];

typedef structATTLookaheadInfo_t
{
            ATTInterflow_t          type;
            ATTPriority_t           priority;
            short                   hours;
            short                   minutes;
            short                   seconds;
            DeviceID_t              sourceVDN;
```

Private Data Version 6 Syntax (Continued)

```
            ATTUnicodeDeviceID_tuSourceVDN; // sourceVDN in Unicode
} ATTLookaheadInfo_t;

typedef enum ATTInterflow_t
{
            LAI_NO_INTERFLOW= -1,   // indicates Info not present
            LAI_ALL_INTERFLOW= 0,
            LAI_THRESHOLD_INTERFLOW= 1,
            LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
            LAI_NOT_IN_QUEUE= 0,
            LAI_LOW                 = 1,
            LAI_MEDIUM              = 2,
            LAI_HIGH                = 3,
            LAI_TOP                 = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
            ATTUserEnteredCodeType_t            type;
            ATTUserEnteredCodeIndicator_t       indicator;
            char                                data[25];
            DeviceID_t                          collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
            UE_NONE                 = -1,
            UE_ANY                  = 0,
            UE_LOGIN_DIGITS         = 2,
            UE_CALL_PROMPTER        = 5,
            UE_DATA_BASE_PROVIDED   = 17,
            UE_TONE_DETECTOR        = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
            UE_COLLECT  = 0,
            UE_ENTERED  = 1
} ATTUserEnteredCodeIndicator_t;
```

Private Data Version 6 Syntax (Continued)

```
#define ATT_MAX_USER_INFO 129
#define ATT_MAX_UUI_SIZE 96
#define ATTV5_MAX_UUI_SIZE 32

typedef struct  ATTUserToUserInfo_t  {
    ATTUUIProtocolType_t type;
    struct {
                short length; // 0 indicates UUI not present
                unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;


typedef enum ATTUUIProtocolType_t
{
            UUI_NONE    = -1,       // indicates not specified
            UUI_USER_SPECIFIC= 0,   // user-specific
            UUI_IA5_ASCII = 4       // null terminated ascii char string
} ATTUUIProtocolType_t;

typedef struct ATTCallOriginatorInfo_t
{
            Boolean     hasInfo;    // if FALSE, no callOriginatorType
            short       callOriginatorType;
} ATTCallOriginatorInfo_t;
```

Private Data Version 5 Syntax

If private data accompanies a CSTARouteRequestExtEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then the CSTARouteRequestExtEvent does not deliver private data to the application.

If acsGetEventBlock() or acsGetEventPoll() returns a Private Data length of 0, then no private data is provided with this Route Request Event.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV5RouteRequestEvent - CSTA Unsolicited Event Private Data

typedef struct
{
             ATTEventType  eventType;    // ATTV5_ROUTE_REQUEST
             union
             {
                          ATTV5RouteRequestEvent_tv5routeRequest;
             } u;
} ATTEvent_t;

typedef struct ATTV5RouteRequestEvent_t
{
             DeviceID_t                 trunkGroup;
             ATTLookaheadInfo_t         lookaheadInfo
             ATTUserEnteredCode_t       userEnteredCode;
             ATTUserToUserInfo_t        userInfo;
             ATTUCID_t                  ucid;
ATTCallOriginatorInfo_t                 callOriginatorInfo;
             Boolean                    flexibleBilling;
} ATTV5RouteRequestEvent_t;

typedef char ATTUCID_t[64];

typedef structATTLookaheadInfo_t
{
             ATTInterflow_t             type;
             ATTPriority_t              priority;
             short                      hours;
             short                      minutes;
             short                      seconds;
             DeviceID_t                 sourceVDN;
             ATTUnicodeDeviceID_tuSourceVDN; // sourceVDN in Unicode
} ATTLookaheadInfo_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef enum ATTInterflow_t
{
            LAI_NO_INTERFLOW= -1,      // indicates Info not present
            LAI_ALL_INTERFLOW          = 0,
            LAI_THRESHOLD_INTERFLOW = 1,
            LAI_VECTORING_INTERFLOW = 2
} ATTInterflow_t;

typedef enum ATTPriority_t
{
            LAI_NOT_IN_QUEUE           = 0,
            LAI_LOW                    = 1,
            LAI_MEDIUM                 = 2,
            LAI_HIGH                   = 3,
            LAI_TOP                    = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
            ATTUserEnteredCodeType_t               type;
            ATTUserEnteredCodeIndicator_t          indicator;
            char                                   data[25];
            DeviceID_t                             collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
            UE_NONE                    = -1,
            UE_ANY                     = 0,
            UE_LOGIN_DIGITS            = 2,
            UE_CALL_PROMPTER           = 5,
            UE_DATA_BASE_PROVIDED      = 17,
            UE_TONE_DETECTOR           = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
            UE_COLLECT    = 0,
            UE_ENTERED    = 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTV5UserToUserInfo_t
{
            ATTUUIProtocolType_t type;
            struct {
            short          length;    // 0 indicates UUI not present
            unsigned char value[33];
            } data;
} ATTV5UserToUserInfo_t;
```

Private Data Version 5 Syntax (Continued)

```
typedef enum ATTUUIProtocolType_t
{
            UUI_NONE    = -1,       // indicates not specified
            UUI_USER_SPECIFIC= 0,   // user-specific
            UUI_IA5_ASCII= 4        // null terminated ascii char string
} ATTUUIProtocolType_t;

typedef struct ATTCallOriginatorInfo_t
{
            Boolean     hasInfo;    // if FALSE, no callOriginatorType
            short       callOriginatorType;
} ATTCallOriginatorInfo_t;
```

Private Data Versions 2-4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV4RouteRequestEvent - CSTA Unsolicited Event Private Data

typedef struct
{
            ATTEventTypeeventType;  // ATTV4_ROUTE_REQUEST
            union
            {
                        ATTV4RouteRequestEvent_tv4routeRequest;
            } u;
} ATTEvent_t;

typedef struct ATTV4RouteRequestEvent_t
{
            DeviceID_t              trunk;
            ATTV4LookaheadInfo_t    lookaheadInfo
            ATTUserEnteredCode_t    userEnteredCode;
            ATTUserToUserInfo_t     userInfo;
} ATTV4RouteRequestEvent_t;

typedef structATTV4LookaheadInfo_t
{
            ATTInterflow_t          type;
            ATTPriority_t           priority;
            short                   hours;
            short                   minutes;
            short                   seconds;
            DeviceID_t              sourceVDN;
} ATTLookaheadInfo_t;

typedef enum ATTInterflow_t
{
            LAI_NO_INTERFLOW= -1,   // indicates Info not present
            LAI_ALL_INTERFLOW= 0,
            LAI_THRESHOLD_INTERFLOW= 1,
            LAI_VECTORING_INTERFLOW= 2
} ATTInterflow_t;
```

Private Data Versions 2-4 Syntax (Continued)

```
typedef enum ATTPriority_t
{
            LAI_NOT_IN_QUEUE        = 0,
            LAI_LOW                 = 1,
            LAI_MEDIUM              = 2,
            LAI_HIGH                = 3,
            LAI_TOP                 = 4
} ATTPriority_t;

typedef struct ATTUserEnteredCode_t
{
            ATTUserEnteredCodeType_t            type;
            ATTUserEnteredCodeIndicator_t       indicator;
            char                                data[25];
            DeviceID_t                          collectVDN;
} ATTUserEnteredCode_t;

typedef enum ATTUserEnteredCodeType_t
{
            UE_NONE                 = -1,
            UE_ANY                  = 0,
            UE_LOGIN_DIGITS         = 2,
            UE_CALL_PROMPTER        = 5,
            UE_DATA_BASE_PROVIDED   = 17,
            UE_TONE_DETECTOR        = 32
} ATTUserEnteredCodeType_t;

typedef enum ATTUserEnteredCodeIndicator_t
{
            UE_COLLECT  = 0,
            UE_ENTERED  = 1
} ATTUserEnteredCodeIndicator_t;

typedef struct ATTUserToUserInfo_t
{
            ATTUUIProtocolType_t    type;
            struct {
            short                   length; // 0 indicates UUI not present
            unsigned char           value[33];
            } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t
{
            UUI_NONE    = -1,       // indicates not specified
            UUI_USER_SPECIFIC= 0,   // user-specific
            UUI_IA5_ASCII = 4       // null terminated ascii char string
} ATTUUIProtocolType_t;
```

Route Request Service (TSAPI Version 1)

Summary

- Direction: Switch to Client
- Event: CSTARouteRequestEvent
- Service Parameters: routeRegisterReqID, routingCrossRefID, currentRoute, callingDevice, routedCall, routedSelAlgorithm, priority, setupInformation

Functional Description:

The switch sends a CSTARouteRequestEvent to request a destination for a call arrived on a routing device from a routing server application. The application may have registered as the routing server for the routing device on the switch that is making the request, or it may have registered as the default routing server. The CSTARouteRequestEvent includes call-related information. A routing server application typically uses the call-related information and a database to determine the destination for the call. The routing server application responds to the CSTARouteRequestExtEvent via a cstaRouteSelect () request that specifies a destination for the call or a cstaRouteEnd () request, if the application has no destination for the call.

Detailed Information:

- The first cstaRouteSelect() response received by the switch is used as the route for the call, and all other CSTARouteRequestEvents for the call are canceled via CSTARouteEndEvents.
- If application terminates the CSTARouteRequestEvent request via a cstaRouteEnd(), the switch continues vector processing.
- A CSTARouteRequestEvent request will not affect the Call Event Reports.

Syntax

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

//  CSTARouteRequestEvent - CSTA Unsolicited Event

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTAREQUEST
            EventType_t eventType;  // CSTA_ROUTE_REQUEST
} ACSEventHeader_t;

typedef long           RouteRegisterReqID_t;

typedef long           RoutingCrossRefID_t;

typedef char            DeviceID_t[64];

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

typedef ExtendedDeviceID_t CallingDeviceID_t;

typedef ExtendedDeviceID_t CalledDeviceID_t;

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

typedef struct ConnectionID_t {
            long                    callID;
            DeviceID_t              deviceID;
            ConnectionID_Device_t   devIDType;
} ConnectionID_t;

typedef enum SelectValue_t {
            SV_NORMAL = 0,
            SV_LEAST_COST = 1,
            SV_EMERGENCY = 2,
            SV_ACD = 3,
            SV_USER_DEFINED = 4
} SelectValue_t;
```

Syntax (Continued)

```
typedef struct SetupValues_t {
            int                     length;
            unsigned char           *value;
} SetupValues_t;

typedef struct {
            RouteRegisterReqID_t    routeRegisterReqID;
            RoutingCrossRefID_t     routingCrossRefID;
            CalledDeviceID_t        currentRoute;
                                    // TSAPI/cstadefs.h is wrong
            CallingDeviceID_t       callingDevice;
                                    // TSAPI/cstadefs.h is wrong
            ConnectionID_t          routedCall;
            SelectValue_t           routedSelAlgorithm;
            Boolean                 priority;
            SetUpValues_t           setupInformation;
} CSTARouteRequestEvent_t;

typedef struct
{
            ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
            InvokeID_t  invokeID;// Unused for Route Request Event
                                    union
                                    {
                        CSTARouteRequestEvent_trouteRequest;
                                    } u;
                        } cstaRequest;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Route Select Service (TSAPI Version 2)

Summary

- Direction: Client to Switch
- Function: cstaRouteSelectInv()
- Private Data Function: *attv7RouteSelect()*, *attV6RouteSelect()* (private data version 6), *attV5RouteSelect()* (private data version 2, 3, 4, and 5)
- Service Parameters: routeRegisterReqID, routingCrossRefID, routeSelected, remainRetry, setupInformation, routeUsedReq
- Private Parameters: callingDevice, directAgentCallSplit, priorityCalling, destRoute, collectCode, userProvidedCode, userInfo, DeviceHistory, Network Call Redirection
- Ack Parameters: noData
- Nak Parameter: universalFailure

Functional Description:

The routing server application uses cstaRouteSelectInv() to provide a destination to the switch in response to a CSTARouteRequestExtEvent for a call.

Service Parameters:

routeRegisterReqID	[mandatory] Contains a handle to the routing registration session for which the application is providing routing service. The routing server application received this handle in a CSTARouteRegisterReqConfEvent confirmation to a cstaRouteRegisterReq() request.
routingCrossRefID	[mandatory] Contains the handle for the routing dialog of this call. The application previously received this handle in the CSTARouteRequestExtEvent for the call.
routeSelected	[mandatory] Specifies a destination for the call. If the destination is an off-PBX number, it can contain the TAC/ARS/AAR information (see destRoute).
remainRetry	[optional - not supported] Specifies the number of times that the application is willing to receive a CSTARouteRequestExtEvent for this call in case the switch needs to request an alternate route.
setupInformation	[optional - not supported] Contains a revised ISDN call setup message that the switch will use to route the call.
routeUsedReq	[optional - supported] Indicates a request to receive a CSTARouteUsedExtEvent for the call. • If specified, the TSAPI Service always returns the same destination information that is specified in the routeSelected and destRoute of this cstaRouteSelectInv().

Private Parameters:

callingDevice [optional] Specifies the calling device. A NULL specifies that this parameter is not present.

directAgentCallSplit [optional] Specifies the ACD agent's split extension for a Direct-Agent call routing. A Direct-Agent call is a special type of ACD call that is directed to a specific agent rather than to any available agent. The agent specified in the routeSelected must be logged into this split. A NULL parameter specifies that this is not a Direct-Agent call.

priorityCalling [mandatory] Specifies the priority of the call. Values are "On" (TRUE) or "Off" (FALSE). When "On" is selected, a priority call is placed if the routeSelected is an on-PBX destination. When "On" is selected for an off-PBX calledDevice, the call will be denied.

destRoute [optional] Specifies the TAC/ARS/AAR information for off-PBX destinations, if the information is not included in the routeSelected. A NULL parameter specifies no TAC/ARS/AAR information.

collectCode [optional] This parameter allows the application to request that a DTMF tone detector (TN744) be connected to the routed call and to detect and collect caller (call originator) entered code/digits.

- These digits are collected while the call is not in vector processing. The switch handles these digits like dial-ahead digits, and they may be used by Call Prompting features. The code/digits collected are passed to the application via event reports.
- The collectParty parameter in collectCode indicates to which party on the call the tone detector should listen. Currently, the call originator is the only option supported.
- The collectCode and userProvidedCode are mutually exclusive. If collectCode is present, then userProvidedCode cannot be present. A NULL indicates this parameter is not specified. If the collectCode type is set to "UC_NONE", it also indicates that no collectCode is sent with this request.

userProvidedCode [optional] This parameter allows the application to send code/digits (ASCII string with 0-9, *, and # only) with the routed call. These code/digits are treated as dial-ahead digits for the call, and are stored in a dial-ahead digit buffer.

- They can be collected (one at a time or in a group) using the collect digits vector command(s) on the switch.
- The userProvidedCode and collectCode parameters are mutually exclusive. If userProvidedCode is present, then collectCode cannot be present.
- A NULL indicates no user provided code. If the userProvidedCode type is set to "UP_NONE", it also indicates no userEnteredCode is sent with this request.

- The # character terminates the Communication Manager collection of user input so it is the last character present in the string if it is sent.

  Note: The user-to-user code collection stops when the user enters the requested number of digits or enters a # character to end the digit entry. If a user enters the # before entering the requested number of digits, then the # appears in the character string.

- Application designers must be aware that if a user enters more digits than requested, the excess digits remain in the Communication Manager prompting buffer and may therefore interfere with any later digit collection or reporting.

userInfo [optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 or 96 bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string.

It is propagated with the call whether the call is routed to a destination on the local switch or to a destination on a remote switch over PRI trunks. The switch sends the user-to-user information (UUI) in the ISDN SETUP message over the PRI trunk to establish the call. The local and the remote switch include the UUI in the Delivered Event Report and in the CSTARouteRequestExtEvent to the application. A NULL indicates that this parameter is not present.

Prior to G3V8, the maximum length of userInfo was 32 bytes. Beginning with G3V8, the maximum length of userInfo was increased to 96 bytes.

An application using private data version 5 and earlier can only receive a maximum of 32-byte data for userInfo, regardless of the size data that is sent by the switch.

The following UUI protocol types are supported:

- UUI_NONE — There is no data provided in the data parameter.
- UUI_USER_SPECIFIC — The content of the data parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the size parameter.
- UUI_IA5_ASCII — The content of the data parameter must be a null-terminated IA5 (ASCII) character string. The correct size (maximum of 32 or 96 bytes excluding the null terminator) of data must be specified in the size parameter.

redirectType This optional parameter specifies whether or not NetworkCallRedirection (NCR) should be invoked. Values are "On" (TRUE) or "Off" (FALSE). When "On" is selected, the routeSelected service parameter specifies a PSTN routing number (without an access code) for NCR requests. If the parameter is not specified, then the value defaults to "Off".

Ack Parameters:

noData None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain one of the following error values, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786. • INVALID_CSTA_DEVICE_IDENTIFIER (12) An invalid routeRegisterReqID has been specified in the Route Select request. • INVALID_CROSS_REF_ID (17) An invalid routeCrossRefID has been specified in the Route Select request.

Detailed Information:

An application may receive one CSTARouteEndEvent and one universalFailure for a cstaRouteSelectInv() request for the same call in one of the following call scenarios:

- Switch/TSAPI Service sends a CSTARouteRequestExtEvent to application.
- Caller drops the call.
- Application sends a CSTARouteSelectInv() request to TSAPI Service.
- Switch/TSAPI Service sends a CSTARouteEndEvent (errorValue = NO_ACTIVE_CALL) to application.
- TSAPI Service receives the CSTARouteSelectInv() request, but call has been dropped.
- TSAPI Service sends universalFailure for the CSTARouteSelectInv() request (errorValue = INVALID_CROSS_REF_ID) to application.

Syntax

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

// cstaRouteSelectInv() - Service Request

RetCode_t cstaRouteSelectInv(
    ACSHandle_t         acsHandle,
    InvokeID_t          invokeID, // V1 & V2 are different here
    RouteRegisterReqID_trouteRegisterReqID,
    RoutingCrossRefID_t routingCrossRefID,
    DeviceID_t          *routeSelected,
    RetryValue_t        remainRetry,
    SetUpValues_t       *setupInformation,
    Boolean             routeUsedReq,
    PrivateData_t       *privateData);

typedef long            RouteRegisterReqID_t;

typedef long            RoutingCrossRefID_t;

typedef char            DeviceID_t[64];

typedef short           RetryValue_t;

typedef struct SetUpValues_t {
            int                     length;
            unsigned char           *value;
} SetUpValues_t;
```

Private Data Version 7 and 8 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

 // attV7RouteSelect() - Service Request Private Data Setup Function

typedef struct ATTRouteSelect_t {
    DeviceID_t      callingDevice;
    DeviceID_t      directAgentCallSplit;
    unsigned char   priorityCalling;
    DeviceID_t      destRoute;
    ATTUserCollectCode_t collectCode;
    ATTUserProvidedCode_t userProvidedCode;
    ATTUserToUserInfo_t userInfo;
    ATTRedirectType_t redirectType;
} ATTRouteSelect_t;
```

Private Data Version 6 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

 // attV6RouteSelect() - Service Request Private Data Setup Function

RetCode_t     attV6RouteSelect(
              ATTPrivateData_t*attPrivateData,        // length must be
                                                      // set
              DeviceID_t    *callingDevice            // not in use
              DeviceID_t    *directAgentCallSplit,    // ACD Agents
                                                      // split
              Boolean       priorityCalling,          // TRUE = On,
                                                      // FALSE = Off
                                                      // (or not
                                                      // specified)
              DeviceID_t    *destRoute,               // TAC/ARS/AAR for
                                                      // off-PBX ext
              ATTUserCollectCode_t*collectCode,       // Request DTMF
                                                      // tone detector
              ATTUserProvidedCode_t*userProvidedCode,// Code to send
                                                      // with routed
                                                      // call
              ATTUserToUserInfo_t*userInfo);          // user-to-user
                                                      // info with call

typedef struct ATTPrivateData_t {
              char                        vendor[32];
              ushort                      length;
              char                        data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;


typedef struct ATTUserCollectCode_t {
              ATTCollectCodeType_ttype;
              short         digitsToBeCollected;  // must be 1 - 24 digits
              short         timeout;      // must be 0 - 63 seconds
              ConnectionID_tcollectParty; // reserved - not in use
                                          // (defaults to call originator)
              ATTSpecificEvent_tspecificEvent;// Ignored (Defaults to Far
                                                      // End Answer)
} ATTUserCollectCode_t;

typedef enum  ATTCollectCodeType_t {
              UC_NONE                     = 0,// indicates UCC not present
              UC_TONE_DETECTOR            = 32
} ATTCollectCodeType_t;

typedef enum  ATTSpecificEvent_t {
              SE_ANSWER                   = 11,
              SE_DISCONNECT= 4
}  ATTSpecificEvent_t;
```

Private Data Version 6 Syntax (Continued)

```
typedef struct ATTUserProvidedCode_t {
ATTProvidedCodeType_t   type;
    char                data[25];
} ATTUserProvidedCode_t;

typedef enum ATTProvidedCodeType_t {
    UP_NONE = 0,        // indicates UPC not present
    UP_DATA_BASE_PROVIDED = 17
} ATTProvidedCodeType_t;

        #define ATT_MAX_USER_INFO 129
        #define ATT_MAX_UUI_SIZE 96
        #define ATTV5_MAX_UUI_SIZE 32

typedef structATTUserToUserInfo_t {
            ATTUUIProtocolType_t type;
            struct {
            short           length; // 0 indicates UUI not present
            unsigned char   value[ATT_MAX_USER_INFO];
            } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
            UUI_NONE= -1            // indicates not specified
            UUI_USER_SPECIFIC= 0,   // user-specific
            UUI_IA5_ASCII= 4        // null terminated ascii
                                    // character string
} ATTUUIProtocolType_t
```

Private Data Version 2-5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

 // attRouteSelect() - Service Request Private Data Setup Function

RetCode_t     attRouteSelect(
              ATTPrivateData_t*attPrivateData,        // length must be
                                                      // set
              DeviceID_t    *callingDevice            // not in use
              DeviceID_t    *directAgentCallSplit,    // ACD Agents
                                                      // split
              Boolean       priorityCalling,          // TRUE = On,
                                                      // FALSE = Off
                                                      // (or not
                                                      // specified)
              DeviceID_t    *destRoute,               // TAC/ARS/AAR for
                                                      // off-PBX ext
              ATTUserCollectCode_t*collectCode,       // Request DTMF
                                                      // tone detector
              ATTUserProvidedCode_t*userProvidedCode,// Code to send
                                                      // with routed
                                                      // call
              ATTUserToUserInfo_t*userInfo);          // user-to-user
                                                      // info with call

typedef struct ATTPrivateData_t {
              char                      vendor[32];
              ushort                    length;
              char                      data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;


typedef struct ATTUserCollectCode_t {
              ATTCollectCodeType_ttype;
              short         digitsToBeCollected;  // must be 1 - 24 digits
              short         timeout;      // must be 0 - 63 seconds
              ConnectionID_tcollectParty; // reserved - not in use
                                        // (defaults to call originator)
              ATTSpecificEvent_tspecificEvent;// Ignored (Defaults to Far
                                                      // End Answer)
} ATTUserCollectCode_t;

typedef enum  ATTCollectCodeType_t {
              UC_NONE                   = 0,// indicates UCC not present
              UC_TONE_DETECTOR          = 32
} ATTCollectCodeType_t;

typedef enum  ATTSpecificEvent_t {
              SE_ANSWER                 = 11,
              SE_DISCONNECT= 4
}  ATTSpecificEvent_t;
```

Private Data Version 2-5 Syntax (Continued)

```
typedef struct ATTUserProvidedCode_t {
ATTProvidedCodeType_t   type;
    char                data[25];
} ATTUserProvidedCode_t;

typedef enum ATTProvidedCodeType_t {
    UP_NONE = 0,        // indicates UPC not present
    UP_DATA_BASE_PROVIDED = 17
} ATTProvidedCodeType_t;

typedef structATTUserToUserInfo_t {
            ATTUUIProtocolType_t type;
            struct {
            short           length; // 0 indicates UUI not present
            unsigned char   value[32];
            } data;
} ATTUserToUserInfo_t;

typedef enum ATTUUIProtocolType_t {
            UUI_NONE= -1            // indicates not specified
            UUI_USER_SPECIFIC= 0,   // user-specific
            UUI_IA5_ASCII= 4        // null terminated ascii
                                    // character string
} ATTUUIProtocolType_t
```

Route Select Service (TSAPI Version 1)

Summary

- Direction: Client to Switch
- Function: cstaRouteSelect()
- Service Parameters: routeRegisterReqID, routingCrossRefID, routeSelected, remainRetry, setupInformation, routeUsedReq

Functional Description:

The routing server application uses cstaRouteSelect () to provide a destination to the switch in response to a CSTARouteRequestEvent for a call.

Detailed Information:

An application may receive two CSTARouteEndEvent(s) for a cstaRouteSelect() request for the same call in one of the following call scenarios:

- Switch/TSAPI Service sends a CSTARouteRequestEvent to application.
- Caller drops the call.
- Application sends a CSTARouteSelect() request to TSAPI Service.
- Switch/TSAPI Service sends a CSTARouteEndEvent (errorValue = NO_ACTIVE_CALL) to application.
- TSAPI Service receives the CSTARouteSelect() request, but call has been dropped.
- TSAPI Service sends a CSTARouteEndEvent for the CSTARouteSelect() request (errorValue = INVALID_CROSS_REF_ID) to application.

Syntax

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

// cstaRouteSelect() - Service Request

RetCode_t     cstaRouteSelect (
    ACSHandle_t         acsHandle,
    RouteRegisterReqID_trouteRegisterReqID,
    RoutingCrossRefID_t routingCrossRefID,
    DeviceID_t          *routeSelected,
    RetryValue_t        remainRetry,
    SetUpValues_t       *setupInformation,
    Boolean             routeUsedReq,
    PrivateData_t       *privateData);

typedef long            RouteRegisterReqID_t;

typedef long            RoutingCrossRefID_t;

typedef char            DeviceID_t[64];

typedef short           RetryValue_t;

typedef struct SetUpValues_t {
            int         length;
            unsigned char*value;
} SetUpValues_t;
```

Route Used Event (TSAPI Version 2)

Summary

- Direction: Switch to Client
- Event: CSTARouteUsedExtEvent
- Private Data Event: ATTRouteUsedEvent
- Service Parameters: routeRegisterReqID, routingCrossRefID, routeUsed, callingDevice, domain
- Private Parameters: destRoute

Functional Description:

The switch uses a CSTARouteUsedExtEvent to provide a destination to the routing server application with the actual destination of a call for which the application previously sent a V containing a destination. The routeUsed and destRoute parameters contain the same information specified in the routeSelected and destRoute parameters of the previous cstaRouteSelectInv() request of this call, respectively. The callingDevice parameter contains the same calling device number provided in the previous CSTARouteRequestExtEvent of this call.

Service Parameters:

routeRegisterReqID	[mandatory] Contains a handle to the routing registration session for which the application is providing routing service. The routing server application received this handle in a CSTARouteRegisterReqConfEvent confirmation to a cstaRouteRegisterReq() request.
routingCrossRefID	[mandatory] Contains the handle for the routing dialog of this call. The application previously received this handle in the CSTARouteRequestExtEvent for the call.
routeUsed	[mandatory] Specifies the destination of the call. This parameter has the same destination specified in the routeSelected of the previous cstaRouteSelectInv() request of this call.
callingDevice	[optional - supported] Specifies the call origination device. It contains the same calling device number provided in the previous CSTARouteRequestExtEvent.
domain	[optional - not supported] Indicates whether the call has left the switching domain accessible to the TSAPI Service. Typically, a call leaves a switching domain when it is routed to a trunk connected to another switch or to the public switch network. This parameter is not supported and is always set to FALSE. This does not mean that the call has (or has not) left Communication Manager. An application should ignore this parameter

Private Parameters:

destRoute	[optional] Specifies the TAC/ARS/AAR information for off-PBX destinations. This parameter contains the same information specified in the destRoute of the previous cstaRouteSelectInv() request of this call.

Detailed Information:

- Note that the number provided in the routeUsed parameter is from the routeSelected parameter of the previous cstaRouteSelectinv() request of this call received by the TSAPI Service. This information in routeUsed is not from Communication Manager and it may not represent the true route that Communication Manager used.
- Note that the number provided in the destRoute parameter is from the destRoute parameter of the previous cstaRouteSelectinv() request of this call received by the TSAPI Service. This information in destRoute is not from the Communication Manager and it may not represent the true route that the Communication Manager used.
- The number provided in the callingDevice parameter is from the callingDevice parameter of the previous CSTARouteRequestExtEvent of this call sent by the TSAPI Service.

Syntax

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

// CSTARouteUsedExtEvent - Route Select Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTAEVENTREPORT
            EventType_t eventType;  // CSTA_ROUTE_USED_EXT
} ACSEventHeader_t;

typedef struct
{
                        ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
                                    union
                                    {
                        CSTARouteUsedExtEvent_trouteUsed;
                                    } u;
                        } cstaEventReport;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTARouteUsedExtEvent_t {
            RouteRegisterReqID_t    routeRegisterReqID;
            RoutingCrossRefID_t     routingCrossRefID;
            DeviceID_t  routeUsed;// V1 & V2 are different here
            DeviceID_t  callingDevice;// TSAPI V1 & V2 are different here
            Boolean     domain;
} CSTARouteUsedExtEvent_t;
```

Route Used Event (TSAPI Version 1)

Summary

- Direction: Switch to Client
- Event: CSTARouteUsedEvent
- Service Parameters: routeRegisterReqID, routingCrossRefID, routeUsed, callingDevice, domain

Functional Description:

The switch uses a CSTARouteUsedExtEvent to provide a destination to the routing server application with the actual destination of a call for which the application previously sent a V containing a destination. The routeUsed and destRoute parameters contain the same information specified in the routeSelected and destRoute parameters of the previous cstaRouteSelectInv() request of this call, respectively. The callingDevice parameter contains the same calling device number provided in the previous CSTARouteRequestExtEvent of this call.

Detailed Information:

- The number provided in the routeUsed parameter is from the routeSelected parameter of the previous cstaRouteSelect() request of this call received by the TSAPI Service.
- The number provided in the callingDevice parameter is from the callingDevice parameter of the previous CSTARouteRequestEvent of this call sent by the TSAPI Service.

Syntax

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

// CSTARouteUsedEvent - Route Select Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTAEVENTREPORT
            EventType_t eventType;  // CSTA_ROUTE_USED
} ACSEventHeader_t;

typedef struct
{
                        ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
                                    union
                                    {
            CSTARouteUsedEvent_trouteUsed;
                                    } u;
                        } cstaEventReport;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTARouteUsedEvent_t {
            RouteRegisterReqID_t    routeRegisterReqID;
            RoutingCrossRefID_t     routingCrossRefID;
            DeviceID_t              routeUsed;
            DeviceID_t              callingDevice;
            Boolean                 domain;
} CSTARouteUsedEvent_t;
```

Private Parameter Syntax

If private data accompanies a CSTARouteUsedExtEvent, then the private data would be stored in the location that the application specified as the privateData parameter in the acsGetEventBlock() or acsGetEventPoll() request. If the privateData pointer is set to NULL in these requests, then CSTARouteUsedExtEvent does not deliver private data to the application.

If the acsGetEventBlock() or acsGetEventPoll() returns Private Data length of 0, then no private data is provided with this Route Request.

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTRouteUsedEvent - Service Response Private Data
typedef struct
{
            ATTEventTypeeventType;  // ATT_ROUTE_USED
            union
            {
                        ATTRouteUsedEvent_tdestRoute;
            }u;
} ATTEvent_t;

typedef struct ATTRouteUsedEvent_t
{
            DeviceID_t    destRoute;
} ATTRouteUsedEvent_t;
```

Chapter 13: System Status Service Group

System Status Services Group describes the services that allow an application to receive reports on the status of the switching system. (System Status services with the driver/switch as the client are not supported.) The following System Status Services and Events are available:

System Status Request Service - cstaSysStatReq()

This service is used by a client application to request system status information from the driver/switch domain.

System Status Start Service - cstaSysStatStart()

This service allows an application to register for System Status event reporting.

System Status Stop Service - cstaSysStatStop()

This service allows an application to cancel a previously registered request for System Status event reporting.

Change System Status Filter Service cstaChangeSysStatFilter()

This service allows an application to request a change in the filter options for System Status event reporting.

System Status Event - CSTASysStatEvent

This unsolicited event informs the application of changes in the system status of the driver/switch.

System Status Events - Not Supported

The following System Status Events are not supported:

- System Status Request Event – CSTASysStatReqEvent
- System Status Request Confirmation – cstaSysStatReqConf()
- System Status Event Send – cstaSysStatEventSend()

System Status Request Service

Summary

- Direction: Client to Switch
- Function: cstaSysStatReq()
- Confirmation Event: CSTASysStatReqConfEvent
- Service Parameters: none
- Ack Parameters: systemStatus
- Ack Private Parameters: count, plinkStatus (private data version 5), linkStatus (private data versions 2, 3, and 4)
- Nak Parameter: universalFailure

Functional Description:

This service is used by a client application to request system status information from the driver/switch.

Service Parameters:

noData	None for this service.

Ack Parameters:

systemStatus [mandatory - partially supported] Provides the application with a cause code defining the overall system status as follows:

NORMAL - This status indication indicates that at least one CTI link to the switch is available. The system status is normal, and TSAPI requests and responses are enabled.

DISABLED - This system status indicates that there is no available CTI link to the switch. The DISABLED status implies that there are no active Monitor requests or Route Register sessions. TSAPI requests and responses are disabled and reject responses should be provided for each request or response.

Ack Private Parameters:

count Identifies the number of CTI links described in the plinkStatus private ack parameter.

plinkStatus Specifies the status of each CTI link to the switch. The TSAPI Service supports multiple CTI links between the Telephony Server and the switch for enhanced throughput and redundancy. The routing of TSAPI service requests and responses over the individual CTI links by the TSAPI Service is hidden from the application.

(The TSAPI application programmer does not need to consider the individual CTI links to a switch when sending/receiving TSAPI service requests/responses.) The plinkStatus private data parameter may be used to check the availability of each administered CTI link to which Communication Manager the application is connected. The status of each link identified by linkID will be set to one of the following values in the linkState field:

- LS_LINK_UP - The link is able to support telephony services to the switch.
- LS_LINK_DOWN - The link is unable to support telephony services to the switch.
- LS_LINK_UNAVAIL - The link has been disabled (busied-out) via the OA&M interface and will not support new CSTA requests. Existing telephony service requests maintained by this link will continue.
- This parameter is supported by private data version 5 and later only.

linkStatus Specifies the status of each CTI link to the switch. For details, see the description for the plinkStatus private ack parameter. This parameter is supported by private data versions 2, 3, and 4.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

Detailed Information:

- Multiple CTI Links - If multiple CTI links are connected and administered to a specific switch, the systemStatus parameter will indicate the aggregate link status. If at least one CTI link is available to support TSAPI requests and responses, the systemStatus will be set to NORMAL. If there are no CTI links to a switch able to support TSAPI requests and responses, the systemStatus will be set to DISABLED.
- If multiple CTI links are connected and administered to a specific switch, Private Data must be used to determine if the switching system is performing as administered. The plinkStatus private parameter can be used to check the status of each individual CTI link.

Syntax

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

// cstaSysStatReq() - Request for system status

RetCode_t  cstaSysStatReq (
             ACSHandle_t               acsHandle,
             InvokeID_t                invokeID,
             PrivateData_t FAR         *privateData);


// CSTASysStatReqConfEvent - System status confirmation event

typedef struct
{
             ACSHandle_t  acsHandle;
             EventClass_t eventClass;  // CSTACONFIRMATION
             EventType_t  eventType;   // CSTA_SYS_STAT_REQ_CONF
} ACSEventHeader_t;

typedef struct
{
             ACSEventHeader_teventHeader;
             union
             {
                          struct
                          {
                          union
                          {
                          CSTASysStatReqConfEvent_t  sysStatReq;
                          } u;
                          } cstaEventReport;
             } event;
             char         heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTASysStatReqConfEvent_t {
             SystemStatus_t            systemStatus;
} CSTASysStatReqConfEvent_t;

typedef enum SystemStatus_t {
    SS_INITIALIZING       = 0,         // Not supported
    SS_ENABLED            = 1,         // Not supported
    SS_NORMAL             = 2,         // Supported
    SS_MESSAGES_LOST      = 3,         // Not supported
    SS_DISABLED           = 4,         // Supported
SS_OVERLOAD_IMMINENT      = 5,         // Not supported
    SS_OVERLOAD_REACHED   = 6,         // Not supported
    SS_OVERLOAD_RELIEVED  = 7          // Not supported
} SystemStatus_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTLinkStatusEvent - Service Response Private Data

typedef struct
{
            ATTEventType_teventType;// ATT_LINK_STATUS
            union
            {
                        ATTLinkStatusEvent_tlinkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTLinkStatusEvent_t
{
            int                     count;
            ATTLinkStatus_tFAR      *pLinkStatus;
} ATTLinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short       linkID;
            ATTLinkState_t linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

Private Data Version 4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV4LinkStatusEvent - Service Response Private Data

typedef struct
{
            ATTEventType_teventType;// ATTV4_LINK_STATUS
            union
            {
                        ATTV4LinkStatusEvent_tv4linkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTV4LinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         linkStatus[8];
} ATTV4LinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short       linkID;
            ATTLinkState_t linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

Private Data Versions 2 and 3 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV3LinkStatusEvent - Service Response Private Data

typedef struct
{
            ATTEventType_teventType;// ATTV3_LINK_STATUS
            union
            {
                        ATTV3LinkStatusEvent_tv3linkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTV3LinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         linkStatus[4];
} ATTV3LinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short       linkID;
            ATTLinkState_t linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

System Status Start Service

Summary

- Direction: Client to Switch
- Function: cstaSysStatStart()
- Confirmation Event: CSTASysStatStartConfEvent
- Private Data Function: attSysStat()
- Service Parameters: statusFilter
- Private Parameters: linkStatReq
- Ack Parameters: statusFilter
- Ack Private Parameters: count, plinkStatus (private data version 5), linkStatus (private data versions 2, 3, and 4)
- Nak Parameter: universalFailure

Functional Description:

This service allows the application to register for System Status event reporting from the driver/switch. The application can register to receive a CSTASysStatEvent each time the status of the driver/switch changes. The service request includes a filter so the application can filter those status events that are not of interest to the application. Only one cstaSysStatStart() request is allowed for an acsOpenStream() request. If one exists, the second one will be rejected.

Service Parameters:

statusFilter	[mandatory - partially supported] A filter used to specify the system status events that are not of interest to the application. If a bit in statusFilter is set to TRUE (1), the corresponding event will not be sent to the application. The only System Status events that will be reported are SS_ENABLED, SS_NORMAL and SS_DISABLED. A request to filter any other System Status events will be ignored.

Private Parameters:

linkStatReq

[optional] The application can use the linkStatReq private parameter to request System Status events for changes in the state of individual G3 switch CTI links. The linkStatReq private parameter is only useful for multilink configurations.

If linkStatReq is set to TRUE (ON), System Status Event Reports will be sent for changes in the states of each individual CTI link. When a CTI link changes between up (LS_LINK_UP), down (LS_LINK_DOWN), or unavailable/ busied-out (LS_LINK_UNAVAIL), a System Status Event Report will be sent to the application. The private data in the System Status Event Report will include the link ID and state for each CTI link to Communication Manager, and not just the link ID and state of the CTI link that experienced a state transition.

If the linkStatReq private parameter was not specified or set to FALSE, changes in the states of individual G3 CTI links will not result in System Status Event Reports unless all links are down, or the first link is established. (The System Status Event Report is always sent when all links are down, or when the first link is established from an "all CTI links down" state.)

Ack Parameters:

statusFilter

[optional - partially supported] Specifies the System Status Event Reports that are to be filtered before they reach the application. The statusFilter may not be the same as the statusFilter specified in the service request, because filters for System Status Events that are not supported are always turned on (TRUE) in systemFilter.

The following filters will always be set to ON, meaning that there are no reports supported for these events:

- SF_INITIALIZING
- SF_MESSAGES_LOST
- SF_OVERLOAD_IMMINENT
- SF_OVERLOAD_REACHED
- SF_OVERLOAD_RELIEVED

Ack Private Parameters:

count	Identifies the number of CTI links described in the plinkStatus private ack parameter. This parameter is only provided when the linkStatusReq private parameter was set to TRUE.
plinkStatus	Specifies the status of each CTI link to the switch. This parameter is only provided when the linkStatusReq private parameter was set to TRUE. The plinkStatus private data parameter will indicate the availability of each administered CTI link to Communication Manager to which the application is connected. The status of each link identified by linkID will be set to one of the following values in the linkState field: ● LS_LINK_UP - The link is able to support telephony services to the switch. ● LS_LINK_DOWN - The link is unable to support telephony services to the switch. ● LS_LINK_UNAVAIL -The link has been disabled (busied-out) via the OA&M interface and will not support new CSTA requests. Existing telephony service requests maintained by this link will continue. ● This parameter is supported by private data version 5 and later only.
linkStatus	Specifies the status of each CTI link to the switch. For details, see the description for the plinkStatus private ack parameter. This parameter is supported by private data versions 2, 3, and 4.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786. ● GENERIC_OPERATION_REJECTION (71)Only one cstaSysStatStart() request is allowed for an acsOpenStream() request. If one exists, the second one will be rejected.

Detailed Information:

- The linkStatReq private parameter is only useful in multilink configurations.
- Only one cstaSysStatStart() request is allowed for an acsOpenStream() request. If one exists, the second one will be rejected. An application can cancel a request for System Status event reporting via cstaSysStatusStop(), and then issue a subsequent cstaSysStatStart() request.
- If the application requests System Status Event Reports for changes in specific CTI link states (up/down/unavailable), it must examine the private data included in the CSTASysStatEvent Event Report to determine the changes in the individual CTI links.

- The count and plinkStatus private ack parameters will only be provided when the linkStatReq parameter was set to TRUE in the System Status Start service request.
- A CSTASysStatEvent event report will be sent with the systemStatus set to SS_DISABLED when the last CTI link to Communication Manager has failed. The application can examine the private data portion of the event report, but it will always indicate that all CTI links are down (LS_LINK_DOWN) or unavailable (LS_LINK_UNAVAILABLE). All Call and Device Monitors will be terminated, all Routing Sessions will be aborted, and all outstanding CSTA requests should be negatively acknowledged.
- A CSTASysStatEvent Event Report will be sent with the systemStatus set to SS_ENABLED when the first CTI link to Communication Manager has been established from an "all CTI links down" state. The application can examine the private data portion of the Event Report to determine which CTI links are up (LS_LINK_UP), which CTI links are down (LS_LINK_DOWN), and which CTI links are disabled via the Telephony Services Administrator interface (LS_LINK_UNAVAIL). No Call or Device Monitors, or Routing Sessions should exist at this point.
- A CSTASysStatEvent Event Report will be sent with the systemStatus set to SS_NORMAL when the application has requested event reports for changes in specific CTI link states (via the linkStatusReq private parameter) and a CTI link changes state to up, (LS_LINK_UP) down (LS_LINK_DOWN), or unavailable/busied-out via OA&M(LS_LINK_UNAVAIL).
- Note that the systemStatus is set to SS_NORMAL, indicating that at least one CTI link to the switch is available. The application can examine the private data portion of the event report to determine which CTI links are up, down, or unavailable/busied-out. Call or Device Monitors, and Routing Sessions may have been terminated when the CTI link state changed to down (LS_LINK_DOWN).

Syntax

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

// cstaSysStatStart() - Service Request

RetCode_t   cstaSysStatStart(
            ACSHandle_t             acsHandle,
            InvokeID_t              invokeID,
            SystemStatusFilter_t    statusFilter,
            PrivateData_tFAR        *privateData);

typedef unsignedSystemStatusFilter_t;

#define     SF_INITIALIZING         0x80        // Not supported
#define     SF_ENABLED              0x40        // Supported
#define     SF_NORMAL               0x20        // Supported
#define     SF_MESSAGES_LOST        0x10        // Not supported
#define     SF_DISABLED             0x08        // Supported
#define     SF_OVERLOAD_IMMINENT    0x04        // Not supported
#define     SF_OVERLOAD_REACHED     0x02        // Not supported
#define     SF_OVERLOAD_RELIEVED    0x01        // Not supported

// CSTASysStatStartConfEvent - Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTACONFIRMATION
            EventType_t eventType;  // CSTA_SYS_STAT_START_CONF
} ACSEventHeader_t;

typedef struct CSTASysStatStartConfEvent_t {
            SystemStatusFilter_t    statusFilter;
} CSTASysStatStartConfEvent_t;
```

Syntax (Continued)

```
typedef struct
{
            ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
                        InvokeID_t  invokeID;
                        union
                                    {
                        CSTASysStatStartConfEvent_t
            sysStatStart;
                                    } u;

                        } cstaConfirmation;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSysStat() - Service Request Private Data Setup Function

RetCode_t    attSysStat(
            ATTPrivateData_tFAR *attPrivateData,// length must be set
            Boolean     linkStatusReq);// send event reports for CTI
                                       // link state changes

typedef struct ATTPrivateData_t
{
            char                    vendor[32];
            unsigned    short       length;
            char                    data[ATT_MAX_PRIVATE_DATA];
}

// ATTLinkStatusEvent - Service Response Private Data

typedef struct
{
            ATTEventTypeeventType;  // ATT_LINK_STATUS
            union
            {
                        ATTLinkStatusEvent_tlinkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTLinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_tFAR      *pLinkStatus;
} ATTLinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short                   linkID;
            ATTLinkState_t          linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

Private Data Version 4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSysStat() - Service Request Private Data Setup Function

RetCode_t    attSysStat(
            ATTPrivateData_t *attPrivateData,// length must be set
            Boolean     linkStatusReq);// send event reports for CTI
                                       // link state changes

typedef struct ATTPrivateData_t
{
            char                    vendor[32];
            unsigned    short       length;
            char                    data[ATT_MAX_PRIVATE_DATA];
}

// ATTV4LinkStatusEvent - Service Response Private Data

typedef struct
{
            ATTEventTypeeventType;  // ATTV4_LINK_STATUS
            union
            {
                        ATTV4LinkStatusEvent_tv4linkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTV4LinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         linkStatus[8];
} ATTV4LinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short                   linkID;
            ATTLinkState_t          linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

Private Data Versions 2 and 3 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSysStat() - Service Request Private Data Setup Function

RetCode_t    attSysStat(
            ATTPrivateData_t *attPrivateData,// length must be set
            Boolean     linkStatusReq);// send event reports for CTI
                                       // link state changes

typedef struct ATTPrivateData_t
{
            char                    vendor[32];
            unsigned    short       length;
            char                    data[ATT_MAX_PRIVATE_DATA];
}

// ATTV3LinkStatusEvent - Service Response Private Data

typedef struct
{
            ATTEventTypeeventType;  // ATTV3_LINK_STATUS
            union
            {
                        ATTV3LinkStatusEvent_tv3linkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTV3LinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         linkStatus[4];
} ATTV3LinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short                   linkID;
            ATTLinkState_t          linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

System Status Stop Service

Summary

- Direction: Client to Switch
- Function: cstaSysStatStop()
- Confirmation Event: CSTASysStatStopConfEvent
- Service Parameters: none
- Ack Parameters: none
- Nak Parameter: universalFailure

Functional Description:

This service allows the application to cancel a previously registered monitor for System Status event reporting from the driver/switch domain

Service Parameters:

noData	None for this service.

Ack Parameters:

noData	None for this service.

Nak Parameter:

universalFailure	If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

Detailed Information:

- An application may receive CSTASysStatEvents from the driver/switch until the CSTASysStatStopConfEvent response is received. The application should check the confirmation event to verify that the System Status monitor has been deactivated.

  After the TSAPI Service has issued the CSTASysStatStopConfEvent, automatic notification of System Status Events will be terminated.

Syntax

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

// cstaSysStatStop() - Service Request

RetCode_t   cstaSysStatStop (
            ACSHandle_t             acsHandle,
            InvokeID_t              invokeID,
            PrivateData_t           *privateData);

// CSTASysStatStopConfEvent - Service Response

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTACONFIRMATION
            EventType_t eventType;  // CSTA_SYS_STAT_STOP_CONF
} ACSEventHeader_t;

typedef char Nulltype;

typedef struct CSTASysStatStopConfEvent_t {
    Nulltype        null;
} CSTASysStatStopConfEvent_t;

typedef struct
{
            ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
                                    InvokeID_t  invokeID;
                                    union
                                    {
                        CSTASysStatStopConfEvent_tsysStatStop;
                                    } u;
                        } cstaConfirmation;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Change System Status Filter Service

Summary

- Direction: Client to Switch
- Function: cstaChangeSysStatFilter()
- Confirmation Event: CSTAChangeSysStatFilterConfEvent
- Private Data Function: attSysStat()
- Service Parameters: statusFilter
- Private Parameters: linkStatReq
- Ack Parameters: statusFilterSelected, statusFilterActive
- Ack Private Parameters: count, plinkStatus (private data version 5), linkStatus (private data versions 2, 3, and 4)
- Nak Parameter: universalFailure

Functional Description:

This service allows the application to modify the filter used for System Status event reporting from the driver/switch domain. The application can filter those System Status events that it does not wish to receive. A CSTASysStatEvent will be sent to the application if the event occurs and the application has not specified a filter for that System Status Event. The application must have previously requested System Status Event reports via the cstaSysStatStart() request, else the cstaChangeSysStatFilter() will be rejected.

Service Parameters:

statusFilter	[mandatory - partially supported] A filter used to specify the System Status Events that are not of interest to the application. If a bit in statusFilter is set to TRUE (1), the corresponding event will not be sent to the application. The only System Status Events that will be reported are SS_ENABLED, SS_NORMAL and SS_DISABLED. A request to filter any other System Status Events will be ignored.

Private Parameters:

linkStatReq [optional] The application can use the linkStatReq private parameter to request System Status Events for changes in the state of individual G3 switch CTI links.

- If linkStatReq is set to TRUE (ON), System Status Event Reports will be sent for changes in the states of each individual CTI link. When a CTI link changes between up (LS_LINK_UP), down (LS_LINK_DOWN), or unavailable/busied-out (LS_LINK_UNAVAIL), a System Status Event Report will be sent to the application. The private data in the System Status Event Report will include the link ID and state for each CTI link to Communication Manager, and not just the link ID and state of the CTI link that experienced a state transition.
- If the linkStatReq private parameter was set to FALSE, changes in the states of individual G3 CTI links will not result in System Status Event Reports unless all links are down, or the first link is established. (The System Status Event Report is always sent when all links are down, or when the first link is established from an "all links down" state.)
- If the linkStatReq private parameter was not specified, there will be no change in the reporting changes in the state of individual G3 CTI links. (If System Status Event Reports were sent for changes in individual G3 CTI links before a cstaChangeStatFilter() service request with no private data, the System Status Event Reports will continue to be sent after the CSTAChangeSysStatFilterConfEvent service response is received, and vice-versa.)

Ack Parameters:

statusFilterSelected [mandatory - partially supported] specifies the System Status Event Reports that are to be filtered before they reach the application. The statusFilterSelected may not be the same as the statusFilter specified in the service request, because filters for System Status Events that are not supported are always turned on in statusFilterSelected. The following filters will always be set to ON, meaning that there are no reports supported for these events:

- SF_INITIALIZING
- SF_MESSAGES_LOST
- SF_OVERLOAD_IMMINENT
- SF_OVERLOAD_REACHED
- SF_OVERLOAD_RELIEVED

statusFilterActive [mandatory - partially supported] Specifies the System Status Event Reports that were already active before the CSTAChangeSysStatConfEvent was issued by the driver. The following filters will always be set to ON, meaning that there are no reports supported for these events:

- SF_INITIALIZING
- SF_MESSAGES_LOST
- SF_OVERLOAD_IMMINENT
- SF_OVERLOAD_REACHED
- SF_OVERLOAD_RELIEVED

Ack Private Parameters:

count Identifies the number of CTI links described in the plinkStatus private ack parameter. This parameter is only provided when the linkStatusReq private parameter was set to TRUE.

plinkStatus Specifies the status of each CTI link to the switch. This parameter is only provided when the linkStatusReq private parameter was set to TRUE. The plinkStatus private data parameter will indicate the availability of each administered CTI link to the G3 to which the application is connected. The status of each link identified by linkID will be set to one of the following values in the linkState field:

- LS_LINK_UP - The link is able to support traffic.
- LS_LINK_DOWN - The link is unable to support traffic.

LS_LINK_UNAVAIL - The link has been disabled (busied-out) via the OA&M interface and will not support new CSTA requests. Existing telephony service requests maintained by this link will continue.

This parameter is supported by private data version 5 and later only.

linkStatus — Specifies the status of each CTI link to the switch. For details, see the description for the plinkStatus private ack parameter. This parameter is supported by private data versions 2, 3, and 4.

Nak Parameter:

universalFailure — If the request is not successful, the application will receive a CSTAUniversalFailureConfEvent. The error parameter in this event may contain the following error value, or one of the error values described in Table 20: Common switch-related CSTA Service errors -- universalFailure on page 786.

GENERIC_OPERATION_REJECTION (71) If the application has not registered to receive System Status Event reports, the cstaChangeSysStatFilter() request will be rejected.

Detailed Information:

- The linkStatReq private parameter is only useful in multilink configurations.
- If the application requests System Status Event Reports for changes in specific CTI link states (up/down/unavailable), they must examine the private data included in the CSTASysStatEvent event report to determine the changes in the individual CTI links.
- The count and plinkStatus private ack parameters will only be provided when the linkStatReq parameter was set to TRUE in the Change System Status Start service request.
- If the linkStatReq private parameter was not specified, there will be no changes in the reporting of System Status events for changes in the state of individual G3 CTI links.
- For more information, refer to System Status Event in this chapter.

Syntax

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

// cstaChangeSysStatFilter() - Service Request

RetCode_t   cstaChangeSysStatFilter (
             ACSHandle_t               acsHandle,
             InvokeID_t                invokeID,
             SystemStatusFilter_t      statusFilter,
             PrivateData_t             *privateData);

typedef unsigned char           SystemStatusFilter_t;

#define                     SF_INITIALIZING          0x80
#define                     SF_ENABLED               0x40
#define                     SF_NORMAL                0x20
#define                     SF_MESSAGES_LOST         0x10
#define                     SF_DISABLED              0x08
#define                     SF_OVERLOAD_IMMINENT     0x04
#define                     SF_OVERLOAD_REACHED      0x02
#define                     SF_OVERLOAD_RELIEVED     0x01

// CSTAChangeSysStatFilterConfEvent - Service Response

typedef struct
{
             ACSHandle_t  acsHandle;
             EventClass_t eventClass;  // CSTACONFIRMATION
             EventType_t  eventType;   // CSTA_CHANGE_SYS_STAT_FILTER_CONF
} ACSEventHeader_t;

typedef struct CSTAChangeSysStatFilterConfEvent_t {
             SystemStatusFilter_t      statusFilterSelected;
             SystemStatusFilter_t      statusFilterActive;
} CSTAChangeSysStatFilterConfEvent_t;

typedef struct
{
             ACSEventHeader_teventHeader;
             union
             {
                          struct
                          {
                                       InvokeID_t   invokeID;
                                       union
                                       {
             CSTAChangeSysStatFilterConfEvent_tchangeSysStatFilter;
                                       } u;
                          } cstaConfirmation;
             } event;
             char         heap[CSTA_MAX_HEAP];
} CSTAEvent_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSysStat() - Service Request Private Data Setup Function

RetCode_t    attSysStat(
            ATTPrivateData_t*attPrivateData, // length must be set
            Boolean     linkStatusReq); // send event reports for
                                        // CTI link state changes

typedef struct ATTPrivateData_t
{
            char                    vendor[32];
            unsigned short          length;
            char                    data[ATT_MAX_PRIVATE_DATA];
}

// ATTLinkStatusEvent - Service Response Private Data

typedef struct
{
            ATTEventType_teventType;// ATT_LINK_STATUS
            union
            {
                        ATTLinkStatusEvent_tlinkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTLinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         *pLinkStatus;
} ATTLinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short                   linkID;
            ATTLinkState_t          linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

Private Data Version 4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSysStat() - Service Request Private Data Setup Function

RetCode_t    attSysStat(
            ATTPrivateData_t*attPrivateData, // length must be set
            Boolean     linkStatusReq); // send event reports for
                                        // CTI link state changes

typedef struct ATTPrivateData_t
{
            char                    vendor[32];
            unsigned short          length;
            char                    data[ATT_MAX_PRIVATE_DATA];
}

// ATTV4LinkStatusEvent - Service Response Private Data

typedef struct
{
            ATTEventType_teventType;// ATTV4_LINK_STATUS
            union
            {
                        ATTV4LinkStatusEvent_tv4linkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTV4LinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         linkStatus[8];
} ATTV4LinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short                   linkID;
            ATTLinkState_t          linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

Private Data Versions 2 and 3 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// attSysStat() - Service Request Private Data Setup Function

RetCode_t    attSysStat(
            ATTPrivateData_t*attPrivateData, // length must be set
            Boolean     linkStatusReq); // send event reports for
                                        // CTI link state changes

typedef struct ATTPrivateData_t
{
            char                    vendor[32];
            unsigned short          length;
            char                    data[ATT_MAX_PRIVATE_DATA];
}

// ATTV3LinkStatusEvent - Service Response Private Data

typedef struct
{
            ATTEventType_teventType;// ATTV3_LINK_STATUS
            union
            {
                        ATTV3LinkStatusEvent_tv3linkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTV3LinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         linkStatus[4];
} ATTV3LinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short                   linkID;
            ATTLinkState_t          linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

System Status Event

Summary

- Direction: Switch to Client
- Event: CSTASysStatEvent
- Service Parameters: systemStatus
- Private Parameters: count, plinkStatus (private data version 5), linkStatus (private data versions 2, 3, and 4)

Functional Description:

This unsolicited event is sent by the G3 driver to inform the application of changes in system status. The application must have previously registered to receive System Status Events via the cstaSysStatStart() service request. The System Status Event Reports will be sent for those events that have not been filtered by the application via the cstaSysStatStart() and cstaChangeSysStatFilter() service requests.

Service Parameters:

systemStatus

[mandatory - partially supported] This parameter contains a value that identifies the change in overall system status detected by the TSAPI Service. The following System Status events will be sent to the application by the G3 driver/switch if the application has not filtered the event:

SS_ENABLED - A CSTASysStatEvent event report will be sent with the systemStatus set to SS_ENABLED when the first CTI link to Communication Manager has been established from an "all CTI links down" state. The application can examine the private data portion of the event report to determine which CTI links are up (LS_LINK_UP), which CTI links are down (LS_LINK_DOWN), and which CTI links are disabled via the OA&M interface (LS_LINK_UNAVAIL). No Call or Device Monitors, or Routing Sessions should exist at this point.

- SS_DISABLED - A CSTASysStatEvent event report will be sent with the systemStatus set to SS_DISABLED when the last CTI link to Communication Manager has failed. The application can examine the private data portion of the event report, but it will always indicate that all CTI links are down (LS_LINK_DOWN) or unavailable (LS_LINK_UNAVAILABLE). All Call and Device Monitors will be terminated, all Routing Sessions will be aborted, and all outstanding CSTA requests should be negatively acknowledged.
- SS_NORMAL- A CSTASysStatEvent event report will be sent with the systemStatus set to SS_NORMAL when the application has requested event reports for changes in specific CTI link states (via the linkStatusReq private parameter in the cstaSysStatStart() or cstaChangeSysStatFilter()) and a CTI link changes state to up, (LS_LINK_UP) down (LS_LINK_DOWN), or unavailable/busied-out via OA&M (LS_LINK_UNAVAIL). The systemStatus normal (SS_NORMAL) indicates that at least one CTI link to the switch is available. The application can examine the private data portion of the event report to determine which CTI links are up, down, or unavailable/busied-out. Call or Device Monitors, and Routing Sessions may have been terminated when the CTI link state changed to down (LS_LINK_DOWN).

Private Parameters

:

count	Identifies the number of CTI links described in the plinkStatus private ack parameter. This parameter is only provided when the linkStatusReq private parameter was set to TRUE.
plinkStatus	Specifies the status of each CTI link to the switch. This parameter is only provided when the linkStatusReq private parameter was set to TRUE. The plinkStatus private data parameter will indicate the availability of each administered CTI link to Communication Manager to which the application is connected. The status of each link identified by linkID will be set to one of the following values in the linkState field: LS_LINK_UP - The link is able to support telephony services to the switch. LS_LINK_DOWN - The link is unable to support telephony services to the switch. LS_LINK_UNAVAIL -The link has been disabled (busied-out) via the OA&M interface and will not support new CSTA requests. Existing telephony service requests maintained by this link will continue. This parameter is supported by private data version 5 and later only.
linkStatus	Specifies the status of each CTI link to the switch. For details, see the description for the plinkStatus private ack parameter. This parameter is supported by private data versions 2, 3, and 4.

Detailed Information:

- If multiple CTI links are connected and administered to a specific switch, the systemStatus parameter will indicate the aggregate link status. When the first CTI link is established from an "all CTI links down" state, a System Status Event Report will be sent to the application with the systemStatus set to SS_ENABLED. When the last CTI fails (a transition to the "all CTI links down" state), a System Status Event Report will be sent to the application with the systemStatus set to SS_DISABLED.
- If multiple CTI links are connected and administered to a specific switch, Private Data must be used to determine if the switching system is performing as administered. The plinkStatus private parameter can be used to check the status of each individual CTI link.

Syntax

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

// CSTASysStatEvent - System Status Event

typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass; // CSTAEVENTREPORT
            EventType_t eventType;  // CSTA_SYS_STAT
} ACSEventHeader_t;

typedef struct
{
                        ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
                                    union
                                    {
                                    CSTASysStatEvent_tsysStat;
                                    } u;
                        } cstaEventReport;
            } event;
            char        heap[CSTA_MAX_HEAP];
} CSTAEvent_t;

typedef struct CSTASysStatEvent_t {
            SystemStatus_t          systemStatus;
} CSTASysStatEvent_t;

typedef enum SystemStatus_t {
    SS_INITIALIZING     = 0,        // Not supported
    SS_ENABLED          = 1,        // Supported
    SS_NORMAL           = 2,        // Supported
    SS_MESSAGES_LOST    = 3,        // Not supported
    SS_DISABLED         = 4,        // Supported
    SS_OVERLOAD_IMMINENT= 5,        // Not supported
    SS_OVERLOAD_REACHED = 6,        // Not supported
    SS_OVERLOAD_RELIEVED= 7         // Not supported
} SystemStatus_t;
```

Private Data Version 5 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTLinkStatusEvent - System Status Event Private Data

typedef struct
{
            ATTEventType_teventType;// ATT_LINK_STATUS
            union
            {
                        ATTLinkStatusEvent_tlinkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTLinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         *pLinkStatus;
} ATTLinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short                   linkID;
            ATTLinkState_t          linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

Private Data Version 4 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV4LinkStatusEvent - System Status Event Private Data

typedef struct
{
            ATTEventType_teventType;// ATTV4_LINK_STATUS
            union
            {
                        ATTV4LinkStatusEvent_tv4linkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTV4LinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         linkStatus[8];
} ATTV4LinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short                   linkID;
            ATTLinkState_t          linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

Private Data Versions 2 and 3 Syntax

```
#include <acs.h>
#include <csta.h>
#include <attpriv.h>

// ATTV3LinkStatusEvent - System Status Event Private Data

typedef struct
{
            ATTEventType_teventType;// ATTV3_LINK_STATUS
            union
            {
                        ATTV3LinkStatusEvent_tv3linkStatus;
            } u;
} ATTEvent_t;

typedef struct ATTV3LinkStatusEvent_t
{
            short                   count;
            ATTLinkStatus_t         linkStatus[4];
} ATTV3LinkStatusEvent_t;

typedef struct ATTLinkStatus_t
{
            short                   linkID;
            ATTLinkState_t          linkState;
} ATTLinkStatus_t;

typedef enum ATTLinkState_t {
            LS_LINK_UNAVAIL= 0,     // the link is disabled
            LS_LINK_UP  = 1,        // the link is up
            LS_LINK_DOWN= 2         // the link is down
} ATTLinkState_t;
```

Appendix A: Universal Failure Events

This appendix contains listings of TSAPI related CSTA messages. It includes the following error listings.

Common switch-related CSTAService errors

Table 20 lists the most commonly used CSTA errors returned by CSTA Services in the CSTAUniversalFailureConfEvent for a negative acknowledgment any CSTA service.

Bear in mind that this table does not include all possible errors. For example, it does not include error codes that are returned by the TSAPI Service. Those error codes are described in “CSTA Data Types,” Chapter 10 of the Application Enablement Services TSAPI Programmer Reference, 02-300545.

WARNING:
An application program should be able to handle any CSTA error defined in the CSTAUniversalFailure_t. Failure to do so may cause the application program to fail.

Because the following errors apply to every CSTA Service supported by the TSAPI Service, they are not repeated for each service description.

Table 20: Common switch-related CSTA Service errors -- *universalFailure*

Error	Description
GENERIC_OPERATION (1)	The CTI protocol has been violated or the service invoked is not consistent with a CTI application association. Report this error -- see Customer Support on page 21.
REQUEST_INCOMPATIBLE_WITH_OBJECT (2)	The service request does not correspond to a CTI application association. Check that the CTI link is administered in Communication Manager with type ADJ-IP. Otherwise, report this error -- Customer Support on page 21.
VALUE_OUT_OF_RANGE (3)	Communication Manager detects that a required parameter is missing in the request or an out-of-range value has been specified.
OBJECT_NOT_KNOWN (4)	The TSAPI Service detects that a required parameter is missing in the request. For example, the deviceID of a connectionID is not specified in a service request.
INVALID_FEATURE (15)	The TSAPI Service detects a CSTA Service request that is not supported by Communication Manager.
GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31)	The request cannot be executed due to a lack of available switch resources.
RESOURCE_OUT_OF_SERVICE (34)	An application can receive this error code when a single CSTA Service request is ending abnormally due to protocol error.
NETWORK_BUSY (35)	Communication Manager is not accepting the request at this time because of processor overload. The application may wish to retry the request but should not do so immediately.
OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44)	The given request cannot be processed due to the system resource limit on the device.
GENERIC_UNSPECIFIED_REJECTION (70)	This is a TSAPI Service internal error, but it cannot be any more specific. A system administrator may find more detailed information about this error in the AE Services OAM error logs. Report this error -- see Customer Support on page 21.
GENERIC_OPERATION_REJECTION (71)	This is a TSAPI Service internal error, but not a defined error. A system administrator should check the TSAPI Service error logs for more detailed information about this error. Report this error -- see Customer Support on page 21.

Table 20: Common switch-related CSTA Service errors -- *universalFailure (continued)*

Error	Description
DUPLICATE_INVOCATION_REJECTION (72)	The TSAPI Service detects that the invokeID in the service request is being used by another outstanding service request. This service request is rejected. The outstanding service request with the same invokeID is still valid.
UNRECOGNIZED_OPERATION_REJECTION (73)	The TSAPI Service detects that the service request from a client application is not defined in the API. A CSTA request with a 0 or negative invokeID will receive this error.
RESOURCE_LIMITATION_REJECTION (75)	The TSAPI Service detects that it lacks internal resources such as the memory or data records to process a service request. A system administrator should check the TSAPI Service error logs for more detailed information about this error. This failure may reflect a temporary situation. The application should retry the request.
ACS_HANDLE_TERMINATION_REJECTION (76)	The TSAPI Service detects that an acsOpenStream session is terminating. The TSAPI Service sends this error for every outstanding CSTA request of this ACS Handle. If the session is not closed in an orderly fashion, the application may not receive this error. For example, a user may power off the PC before the application issues an acsCloseStream request and waits for the confirmation event. In this case, the acsCloseStream is issued by the TSAPI Service on behalf of the application and there is no application to receive this error. If an application issues an acsCloseStream request and waits for its confirmation event, the application will receive this error for every outstanding request.
SERVICE_TERMINATION_REJECTION (77)	The TSAPI Service detects that it cannot provide the service due to the failure or shutting down of the communication link between the Telephony Server and Communication Manager. The TSAPI Service sends this error for every outstanding CSTA request for every ACS Handle affected. Although the link is down or Communication Manager is out of service, the TSAPI Service remains loaded and advertised. When the TSAPI Service is in this state, all CSTA Service requests from a client will receive a negative acknowledgment with this unique error code.

Table 20: Common switch-related CSTA Service errors -- *universalFailure (continued)*

Error	Description
REQUEST_TIMEOUT_REJECTION (78)	The TSAPI Service did not receive the response of a service request sent to Communication Manager more than 20 seconds ago. The timer of the request has expired. The request is canceled and negatively acknowledged with this unique error code. When this occurs, the communication link between the TSAPI Service and Communication Manager may be congested. This can happen when the TSAPI Service exceeds its capacity.
REQUESTS_ON_DEVICE_EXCEEDED_REJECTION (79)	For a device, the TSAPI Service processes one service request at a time. The TSAPI Service queues CSTA requests for a device. Only a limited number of CSTA requests can be queued on a device. Report this error -- see Customer Support on page 21. If this number is exceeded, the incoming client request is negatively acknowledged with this unique error code. Usually an application sends one request and waits for its completion before it makes another request. The MAX_REQS_PER_DEVICE parameter has no effect on this kind of operation. Situations of sending a sequence of requests without waiting for their completion are rare. However, if this is the case, the MAX_REQS_PER_DEVICE parameter should be set to a proper value. The default value for MAX_REQS_PER_DEVICE is 4.

Syntax

The following structure shows only the relevant portions of the unions for this message:

```
typedef struct
{
            ACSHandle_t acsHandle;
            EventClass_teventClass;
            EventType_t eventType;
} ACSEventHeader_t;

typedef struct
{
                        ACSEventHeader_teventHeader;
            union
            {
                        struct
                        {
                                    InvokeID_t  invokeID;
                                    union
                                    {
            CSTAUniversalFailureConfEvent_tuniversalFailure;
                                    } u;
                        }  cstaConfirmation;
            } event;
} CSTAEvent_t;

typedef struct CSTAUniversalFailureConfEvent_t {
    CSTAUniversalFailure_t error;
} CSTAUniversalFailureConfEvent_t;
```

The universalFailure error codes are listed below:

```
typedef enum CSTAUniversalFailure_t {
GENERIC_UNSPECIFIED = 0,
            GENERIC_OPERATION = 1,
            REQUEST_INCOMPATIBLE_WITH_OBJECT = 2,
            VALUE_OUT_OF_RANGE = 3,
            OBJECT_NOT_KNOWN = 4,
            INVALID_CALLING_DEVICE = 5,
            INVALID_CALLED_DEVICE = 6,
            INVALID_FORWARDING_DESTINATION = 7,
            PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE = 8,
            PRIVILEGE_VIOLATION_ON_CALLED_DEVICE = 9,
            PRIVILEGE_VIOLATION_ON_CALLING_DEVICE = 10,
            INVALID_CSTA_CALL_IDENTIFIER = 11,
            INVALID_CSTA_DEVICE_IDENTIFIER = 12,
            INVALID_CSTA_CONNECTION_IDENTIFIER = 13,
            INVALID_DESTINATION = 14,
            INVALID_FEATURE = 15,
```

Syntax (Continued)

```
                INVALID_ALLOCATION_STATE = 16,
                INVALID_CROSS_REF_ID = 17,
                INVALID_OBJECT_TYPE = 18,
                SECURITY_VIOLATION = 19,

        GENERIC_STATE_INCOMPATIBILITY = 21,
                INVALID_OBJECT_STATE = 22,
                INVALID_CONNECTION_ID_FOR_ACTIVE_CALL = 23,
                NO_ACTIVE_CALL = 24,
                NO_HELD_CALL = 25,
                NO_CALL_TO_CLEAR = 26,
                NO_CONNECTION_TO_CLEAR = 27,
                NO_CALL_TO_ANSWER = 28,
                NO_CALL_TO_COMPLETE = 29,
                GENERIC_SYSTEM_RESOURCE_AVAILABILITY = 31,
                SERVICE_BUSY = 32,
                RESOURCE_BUSY = 33,
                RESOURCE_OUT_OF_SERVICE = 34,
                NETWORK_BUSY = 35,
                NETWORK_OUT_OF_SERVICE = 36,
                OVERALL_MONITOR_LIMIT_EXCEEDED = 37,
                CONFERENCE_MEMBER_LIMIT_EXCEEDED = 38,
                GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY = 41,
                OBJECT_MONITOR_LIMIT_EXCEEDED = 42,
                EXTERNAL_TRUNK_LIMIT_EXCEEDED = 43,
                OUTSTANDING_REQUEST_LIMIT_EXCEEDED = 44,
                GENERIC_PERFORMANCE_MANAGEMENT = 51,
                PERFORMANCE_LIMIT_EXCEEDED = 52,
                SEQUENCE_NUMBER_VIOLATED = 61,
                TIME_STAMP_VIOLATED = 62,
                PAC_VIOLATED = 63,
                SEAL_VIOLATED = 64,        // The above errors are detected
                                           // either by the switch or by
                                           // the G3PD.

                                           // The following rejections are
                                           // generated by the G3PD, not by
                                           // the switch.
        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
        } CSTAUniversalFailure_t;
```

Table 21 describes TSLIB error codes. The first column provides the number identifying the error. The second column provides a description of the error. The third column provides possible corrective action for the error or indicates a contact to help you determine the problem.

Table 21: TSLIB Error Codes

Error	Description	Corrective Action
-1	The API version requested is not supported by the existing API client library.	This is an application error; contact the application developer.
-2	One or more of the parameters is invalid.	This is an application error; contact the application developer.
-5	This error code indicates the requested server is not present in the network.	Is the server up? Was the wrong server name used? Are physical connections (wiring) intact?
-6	This return value indicates that there are insufficient resources to open an connection.	Is the correct version of IPX being used? If yes, contact the application developer since the application is trying to open too many connections or is opening streams but not closing them.
-7	The user buffer size was smaller than the size of the next available event.	This is an application error; contact the application developer.
-8	Following initial connection, the server has failed to respond within a specified amount of time (typically 10 seconds)	Call Customer Support and report this error. See Customer Support on page 21.
-9	The connection has encountered an unspecified error.	This is typically a version mismatch. Has some software been replaced or upgraded recently? Call Customer Support and report this error. See Customer Support on page 21.
-10	The ACS handle is invalid.	This is an application error; contact the application developer.
-11	The connection has failed due to network problems. No further operations are possible on this stream. A connection has been lost.	Check see if the TSAPI Service is running. In OAM select **Status and Control** > **Services Summary**. From the Services summary page select **TSAPI Service** and click **Details**. Also, check to see if there is physical connectivity.
-12	Not enough buffers were available to place an outgoing message on the send queue. No message has been sent. This could be either an application error or an indication that the TSAPI Service is overloaded.	In CTI OAM administration, select **Status and Control** > **Switch Connections Summary** to check Tlink resources. If traffic reports show no overload, consult the application developer.

Table 21: TSLIB Error Codes (continued)

Error	Description	Corrective Action
-13	The send queue is full. No message has been sent. This could be either an application error or an indication that the TSAPI Service is overloaded.	In AE Services OAM, select **CTI OAM** > **Status and Control** > **Switch Connections Summary** to check traffic. If traffic reports show no overload, consult the application developer.
-14	This return value indicates that a secure connection could not be opened because there was a problem initializing the Open-SSL library.	
-15	This return value indicates that a stream could not be opened because there was a problem establishing an SSL connection to the server. It may be that the server failed to provide a certificate, or that the server certificate is not signed by a trusted Certificate Authority.	
-16	This return value indicates that a stream could not be opened because the FQDN in the server certificate does not match the expected FQDN.	

ACSUniversalFailureConfEvent error values

Error values in this category indicate that the TSAPI Service detected an ACS-related error. Table 22 describes ACS Universal Failure event error codes.

Note:
The ACSUniversalFailureConfEvent does not indicate a failure or loss of the ACS Stream with the TSAPI Service. If the ACS Stream has failed, then an ACSUniversalFailureEvent (unsolicited version of this confirmation event) is sent to the application, see ACSUniversalFailureEvent on page 110

Table 22: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
0	TSERVER_STREAM_FAILED	TSAPI Service	The client library detected that the connection failed.	1. Other errors may have been sent by the TSAPI Service before the connection was taken down. If so, follow the procedures for this error. 2. If no other errors were received from the TSAPI Service first, then verify that the TSAPI Service/AE Server is still running and look for LAN problems.
1	TSERVER_NO_THREAD	TSAPI Service	The TSAPI Service could not begin execution of a thread group which is necessary for it to run properly.	There is a serious system problem. These errors will appear in the TSAPI Service error logs. Consult the logs for the return code.
2	TSERVER_BAD_DRIVER_ID	TSAPI Service	The TSAPI Service has an internal system error.	This error should never be returned to an application or appear in the TSAPI Service error logs. If this event is generated by the TSAPI Service, then there is a software problem with the TSAPI Service. Call Customer Support and report this error. See Customer Support on page 21.
3	TSERVER_DEAD_DRIVER	TSAPI Service	The specified driver has not sent any heart beat messages to the TSAPI Service for the last three minutes. The driver may be in an inoperable state.	Look for driver error messages and/or contact the driver vendor to determine why it is no longer sending the heartbeat messages.
4	TSERVER_MESSAGE_HIGH_WATER_MARK	TSAPI Service	Obsolete message.	Obsolete message.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
5	TSERVER_FREE_BUFFER_FAILED	TSAPI Service	The TSAPI Service was unable to release TSAPI Service driver interface (TSDI) memory back to the operating system.	Consult the error log files for a corresponding error message. The error code associated with this error message should be one of the following: ● -1 A corresponding FATAL error will be generated indicating the call failed. Follow the description for this error message. ● -2, -9, or -10 Internal TSAPI Service software error. Collect the error log files and message trace files and escalate the problem
6	TSERVER_SEND_TO_DRIVER	TSAPI Service	The TSAPI Service was unable to send a message to the G3PD.	Consult the error log files for a corresponding error message. ● This error can indicate that the driver unregistered while the TSAPI Service was processing messages for it or that there is a software problem with the TSAPI Service. Verify that the driver was loaded at the time of the error. ● The error code (rc) should be one of the following: -2, -6, -9, -10. All these errors indicate an internal TSAPI Service software error. Collect the error log files and message trace files and escalate the problem.
7	TSERVER_RECEIVE_FROM_DRIVER	TSAPI Service	The TSAPI Service was unable to receive a message from the G3PD.	Consult the error log files for a corresponding error message. The error code (rc) should be one of the following: ● -1 A corresponding FATAL error will be generated indicating the call failed. Follow the description for this error message. ● -2 Internal TSAPI Service software error. Collect the error log files and message trace files and escalate the problem.
8	TSERVER_REGISTRATION_FAILED	TSAPI Service	The G3PD, which is internal to the TSAPI Service, failed to register properly. The TSAPI Service will not run properly without this driver.	There is a serious system problem. These errors will appear in the TSAPI Service error logs. Consult the logs for the return code.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
9	TSERVER_SPX_FAILED	TSAPI Service	Obsolete message.	Obsolete message.
10	TSERVER_TRACE	TSAPI Service	This error code has multiple meanings and should not be returned to the application.	Consult the error log files for a corresponding error message.
11	TSERVER_NO_MEMORY	TSAPI Service	The TSAPI Service was unable to allocate a piece of memory.	1. Verify that the server has enough memory to run the TSAPI Service. 2. If the server has enough memory, then the driver has reached its limit of how much memory the TSAPI Service will allocate. This limit is chosen by the driver when it registers with the TSAPI Service. Call Customer Support and report this error. See Customer Support on page 21.
12	TSERVER_ENCODE_FAILED	TSAPI Service	The TSAPI Service was unable to encode a message from the G3PD.	This error should never be returned to an application. Consult the error log files for a corresponding error message. If the error appears in the error logs, it indicates that the TSAPI Service does not recognize the message from the G3PD. Call Customer Support and report this error. See Customer Support on page 21.
13	TSERVER_DECODE_FAILED	TSAPI Service	The TSAPI Service was unable to decode a message from a client workstation.	The application is most likely using an old version of the client library. Check the version to ensure that it supports this message. If you have the latest DLL. Call Customer Support and report this error. See Customer Support on page 21.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
14	TSERVER_BAD_CONNECTION	TSAPI Service	The TSAPI Service tried to process a request with a bad client connection ID number.	This error should never be returned to an application. If it appears in the TSAPI Service error logs, it indicates one of the following: ● an application may have been terminated ● the client workstation was disconnected from the network while the TSAPI Service was processing messages for it. Determine if either of these two cases is true. If this error occurs repeatedly and these conditions are not true, Call Customer Support and report this error. See Customer Support on page 21.
15	TSERVER_BAD_PDU	TSAPI Service	The TSAPI Service received a message from the client that is not a valid TSAPI request.	Verify that the message the client is sending is a valid TSAPI request. If it is then there is a problem with the TSAPI Service. Contact Customer Support (see Customer Support on page 21).
16	TSERVER_NO_VERSION	TSAPI Service	The TSAPI Service received an ACSOpenStreamConfEvent from a driver which does not have one of the version fields set correctly. The confirmation even will still be sent to the client with the version field set to “UNKNOWN.”	This error will appear in the error log files and will indicate which field is invalid. Contact Customer Support (see Customer Support on page 21).
17	TSERVER_ECB_MAX_EXCEEDED	TSAPI Service	Obsolete message.	Obsolete message.
18	TSERVER_NO_ECBS	TSAPI Service	Obsolete message.	Obsolete message.
19	TSERVER_NO_SDB	SDB	The TSAPI Service was unable to initialize the Security Database when loading.	Look for other errors that might indicate a data base initialization problem.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
20	TSERVER_NO_SDB_CHECK_NEEDED	SDB	The TSAPI Service determined that a particular TSAPI message did not require Security Database validation. This code is an internal one and should never be returned to an application.	This error should never be returned to an application or appear in the TSAPI Service error logs. If this event is generated by the TSAPI Service, then there is a software problem with the TSAPI Service. Call Customer Support and report this error. See Customer Support on page 21.
21	TSERVER_SDB_CHECK_NEEDED	SDB	The TSAPI Service determined that a particular TSAPI message did require a Security Database validation. This code is an internal one and should never be returned to an application.	This error should never be returned to an application or appear in the TSAPI Service error logs. If this event is generated by the TSAPI Service, then there is a software problem with the TSAPI Service. Call Customer Support and report this error. See Customer Support on page 21.
22	TSERVER_BAD_SDB_LEVEL	SDB	The TSAPI Service's internal table of API calls indicating which level of security to perform on a specific request is corrupted.	This error should never be returned to an application or appear in the TSAPI Service error logs. If this event is generated by the TSAPI Service, then there is a software problem with the TSAPI Service. Call Customer Support and report this error. See Customer Support on page 21.
23	TSERVER_BAD_SERVERID	SDB	The TSAPI Service rejected an ACSOpenStream request because the server ID in the message did not match a Tlink supported by TSAPI Service.	A software problem has occurred with the application or the client library. Use TS Spy to verify that the application is attempting to open a stream to the correct Tlink.
24	TSERVER_BAD_STREAM_TYPE	SDB	The stream type of an ACSOpenStream request was invalid.	A software problem has occurred with the client library. Call Customer Support and report this error. See Customer Support on page 21.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
25	TSERVER_BAD_PASSWORD_OR_LOGIN	SDB	The password, login, or both from an ACSOpenStream request did not pass the TSAPI Service authentication checks. For more information see "Alternative AE Services Authentication Methods," Chapter 5, of the AE Services Administration and Maintenance Guide (02-300357)	1. Validate that the user login and password were entered correctly into the application. 2. Verify that the user's login and password are correct. 3. If the user must change their password at next login, log in and change the password before starting the application.
26	TSERVER_NO_USER_RECORD	SDB	No user object was found in the security database for the login specified in the ACSOpenStream request.	Verify the user has a user object in the security database by using the CTI OAM. ● Validate that the user's login in the security database exactly matches the Windows username. Create a user object for this user if none exists.
27	TSERVER_NO_DEVICE_RECORD	SDB	No device object was found in the security database for the device specified in the API call.	Create a device object for the device the user is trying to control in the TSAPI Service security database by using the AE Services Operations Administration and Maintenance Web pages (**CTI OAM > Administration > Security Database > Devices**) **Note:** Make sure the assigned Tlink group for this device includes the correct Tlink.
28	TSERVER_DEVICE_NOT_ON_LIST	SDB	The specified device did not appear on any of the searched lists, and more than one of the lists was not blank.	Change the user's administration so that the user has permission to control the device through either the user's worktop object (worktop administration) or through one of the Access Rights (user administration).
30	TSERVER_USERS_RESRTICTED_HOME	SDB	The user tried to access a worktop other than his/her own worktop while the "Extended Worktop Access" feature was disabled; however, permission to access this device on this worktop was granted.	Either enable the "Extended Worktop Access" feature or change the user's worktop or Access Rights options to include permissions for the device at the worktop where the user is logged in.
31	TSERVER_NOAWAYPERMISSION	SDB	Obsolete message.	Obsolete message.
32	TSERVER_NOHOMEPERMISSION	SDB	Obsolete message.	Obsolete message.
33	TSERVER_NOAWAYWORKTOP	SDB	Obsolete message.	Obsolete message.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
34	TSERVER_BAD_DEVICE_RECORD	SDB	The TSAPI Service read a device object from the security database that contained corrupted information. The device object did not contain a PBX index value which is a violation of the SDB structure.	This error should never be returned to an application or appear in the TSAPI Service error logs. If this event is generated by the TSAPI Service, then there is a software problem with the TSAPI Service. Call Customer Support and report this error. See Customer Support on page 21.
35	TSERVER_DEVICE_NOT_SUPPORTED	SDB	The Tlink group administered for this device does not contain the CTI link to which the user opened an connection.	1. Validate that the user opened the connection to the correct CTI link. 2. If the CTI link to which the stream was opened can support this device, use AE Services Operations Administration and Maintenance Web pages (**CTI OAM > Administration > Security Database > Tlinks**) to ensure that the correct Tlink group is assigned to the device or change the Tlink group for the device to “Any Tlink.”
36	TSERVER_INSUFFICIENT_PERMISSION	SDB	Obsolete message.	Obsolete message.
37	TSERVER_NO_RESOURCE_TAG	TSAPI Service	A memory allocation call failed in the TSAPI Service.	There is a serious system problem. These errors will appear in the TSAPI Service error logs. Consult the logs for the return code.
38	TSERVER_INVALID_MESSAGE	TSAPI Service	The TSAPI Service has received a message from the application or the driver that it does not recognize.	Verify that the offending message is valid according to TSAPI. If it is a valid message then there may be a software problem with the TSAPI Service. Call Customer Support and report this error. See Customer Support on page 21.
39	TSERVER_EXCEPTION_LIST	SDB	The device in the API call is a member of an exception group which is administered as part of the user’s worktop, Access Rights, or “Extended Worktop Access” is enabled and the user is logged in.	Determine which of the device groups is an exception group and either remove this device from the group or create a new group that reflects the correct access permissions.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
40	TSERVER_NOT_ON_OAM_LIST	TSAPI Service	The user login which is attempting to open an OAM stream to a PBX driver is a member of an Admin Access group, but this Admin Access group does not contain the OAM Tlink specified by the application.	Use use AE Services Operations Administration and Maintenance Web pages (**CTI OAM > Administration > Security Database > Tlinks**) to verify that this OAM Tlink is assigned to the user's Admin Access Group.
41	TSERVER_PBXID_NOT_IN_SDB	TSAPI Service	An attempt to open a stream to an OAM application was made but the specified Tlink is not in the security database.	1. Verify that the user has entered the correct Tlink name. 2. Use the use AE Services Operations Administration and Maintenance Web pages (**CTI OAM > Administration > Security Database > Tlinks**) to verify that the specified Tlink is registered.
42	TSERVER_USER_LICENSES_EXCEEDED	TSAPI Service	Obsolete message.	Obsolete message.
43	TSERVER_OAM_DROP_CONNECTION	TSAPI Service	The TSAPI Service was used to drop the connection for this client.	Determine why the TSAPI Service administrator dropped the client connection.
44	TSERVER_NO_VERSION_RECORD	TSAPI Service	The TSAPI Service could not find a version stamp on the security database files.	There is a serious problem with the files that make up the TSAPI Service security database. Call Customer Support and report this error. See Customer Support on page 21.
45	TSERVER_OLD_VERSION_RECORD	TSAPI Service	The TSAPI Service found old, out of date version stamps on the security database files.	There is a serious problem with the files that make up the TSAPI Service security database. Call Customer Support and report this error. See Customer Support on page 21.
46	TSERVER_BAD_PACKET	TSAPI Service	Obsolete message.	Obsolete message.
47	TSERVER_OPEN_FAILED	TSAPI Service	The TSAPI Service rejected a user's request to open an connection, so the connection was dropped.	An error code should have been returned in response to the ACSOpenStream() request in the ACSUniversalFailureConfEvent. Follow the procedures defined for that error code.
48	TSERVER_OAM_IN_USE	TSAPI Service	Obsolete message.	Obsolete message.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
49	TSERVER_DEVICE_NOT_ON_HOME_LIST	SDB	The TSAPI Service rejected a user's request to control a device because all of the following are true: ● The Primary Device ID of the user's Worktop does not match the device and the device is not a member of the Secondary Device Group of the user's Worktop. ● The Access Group in the "Access Rights" administration in this user's record which corresponds to the action being attempted (Call Control or Device/Device Monitoring) is empty. ● The "Extended Worktop Access" feature is enabled and the user is not working from his or her own worktop, and either the other worktop is not in the SDB or does not have any devices associated with it.	Grant this user permission to control the device through either of the following ways: ● Edit the worktop object (**CTI OAM > Administration > Security Database > Worktop**) ● Edit the user's "Access Rights" (**CTI OAM > Administration > Security Database > CTI User > Edit CTI User**).
50	TSERVER_DEVICE_NOT_ON_CALL_CONTROL_LIST	SDB	The telephony server rejected a user's request to control a device because all of the following are true: ● There is no worktop or the user has no devices associated with the worktop. ● The "Extended Worktop Access" feature is enabled and the user is not working from his or her own worktop, and either the other worktop is not in the SDB or does not have any devices associated with it.	Change the user's administration so that the user has permission to control the device through either the worktop object (worktop administration) or through the Call Control Access Group "Access Rights" (user administration).

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
51	TSERVER_DEVICE_NOT_ON_AWAY_LIST	SDB	The telephony server rejected a user's request to control a device because all of the following are true: ● There is no worktop or the user has no devices associated with the worktop. ● The Access Group in the "Access Rights" administration in this user's record which corresponds to the action being attempted (Call Control or Device/Device Monitoring) is empty. ● The "Extended Worktop Access" feature is enabled and the user is not working from his or her own worktop, and either the other worktop is not in the SDB or does not have any devices associated with it.	Change the user's administration so that the user has permission to control the device through either the user's worktop object (worktop administration) or through one of the "Access Rights" (user administration).
52	TSERVER_DEVICE_NOT_ON_ROUTE_LIST	SDB	The telephony server has rejected a user's routing request for a device because the user has a routing access group in their Access Rights but the device is not a member of that group.	Change the user's administration so that the user has permission to control the device through the Routing Access Group "Access Rights " (User administration).
53	TSERVER_DEVICE_NOT_ON_MONITOR_DEVICE_LIST	SDB	The telephony server rejected a user's monitor device request because the user has a device/device monitoring access group, but the device is not a member of that group.	Change the user's administration so that the user has permission to control the device through either the worktop record (worktop administration) or through the Device/Device Monitoring Access Group "Access Rights" (User administration).
54	TSERVER_NOT_ON_MONITOR_CALL_DEVICE_LIST	SDB	The telephony server rejected a user's request to monitor a device because the device does not appear on the user's call/device monitor list and the call/device monitor list is not blank.	Change the user's administration so that the user has permission to control the device through the Call/Device Monitoring Access Group "Access Rights" (user administration).
55	TSERVER_NO_CALL_CALL_MONITOR_PERMISSION	SDB	The telephony server rejected a user's request to monitor a device because the Allow option for Call/Call Monitoring Access Group in the "Access Rights" administration in this user's record is disabled.	Enable the Allow option for Call/ Call Monitoring Access Group in the "Access Rights" administration in this user's record (user administration).
56	TSERVER_HOME_DEVICE_LIST_EMPTY	SDB	Obsolete message.	

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
57	TSERVER_CALL_CONTROL_LIST_EMPTY	SDB	Obsolete message.	Obsolete message.
58	TSERVER_AWAY_LIST_EMPTY	SDB	Obsolete message.	Obsolete message.
59	TSERVER_ROUTE_LIST_EMPTY	SDB	The telephony server rejected a user's request to control a device because the "Routing Access Group" in the "Access Rights" administration in this user's record is empty.	Change the user's administration so that the user has permission to control the device through the "Routing Access Group" in "Access Rights" (user administration) by specifying a Device Group for the Routing Access Group.
60	TSERVER_MONITOR_DEVICE_LIST_EMPTY	SDB	Obsolete message.	Obsolete message.
61	TSERVER_MONITOR_CALL_DEVICE_LIST_EMPTY	SDB	The telephony server rejected a user's request to control a device because the Call/Device Monitoring Access Group in the "Access Rights" administration in this user's record is empty.	Change the user's administration so that the user has permission to control the device through the Call/Device Monitoring Access Group under "Access Rights" (user administration).
62	TSERVER_USER_AT_HOME_WORKTOP	SDB	Obsolete message.	Obsolete message.
63	TSERVER_DEVICE_LIST_EMPTY	SDB	All the device groups in a user's worktop and Access Rights are empty (in the set of lists searched for this type of message).	Change the user's administration so that the user has permission to control the device through either the user's worktop record (worktop administration) or through one of the "Access Rights" (user administration).
64	TSERVER_BAD_GET_DEVICE_LEVEL	SDB	A CSTAGetDeviceList query was made with a bad CSTALevel_t value. Valid CSTALevels are: CSTA_HOME_WORK_TOP 1 CSTA_AWAY_WORK_TOP 2 CSTA_DEVICE_DEVICE_MONITOR 3 CSTA_CALL_DEVICE_MONITOR 4 CSTA_CALL_CONTROL 5 CSTA_ROUTING 6	The application has called CSTAGetDeviceList with an invalid device level. Consult the application developer.
65	TSERVER_DRIVER_UNREGISTERED	SDB	The connection was torn down because the PBX driver associated with this stream terminated and unregistered with the TSAPI Service.	Verify that the driver unregistered. If it did not, call Customer Support and report this error. See Customer Support on page 21.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
66	TSERVER_NO_ACS_STREAM	TSAPI Service	The TSAPI Service has received a message from the client or the Tlink over a stream which has not been confirmed. The Tlink may have rejected the ACSOpenStream request or violated the protocol by not returning an ACSOpenStreamConfEvent.	1. The TSAPI Service will terminate this stream when this error occurs. Verify that the application waits for an ACSOpenStreamConfEvent before it makes any further requests. 2. If the application is written correctly, call Customer Support and report this error. See Customer Support on page 21.
67	TSERVER_DROP_OAM	TSAPI Service	Obsolete message.	Obsolete message.
68	TSERVER_ECB_TIMEOUT	TSAPI Service	Obsolete message.	Obsolete message.
69	TSERVER_BAD_ECB	TSAPI Service	Obsolete message.	Obsolete message.
70	TSERVER_ADVERTISE_FAILED	TSAPI Service	The TSAPI Service cannot perform service advertising due to a error.	There is a serious system problem. These errors will appear in the TSAPI Service error logs. Consult the logs for the return code. Call Customer Support and report this error. See Customer Support on page 21.
71	TSERVER_ADVERTISE_FAILED	TSAPI Service	The TSAPI Service encountered an error while trying to access its message trace files or the traffic log.	This a system problem. These errors will appear in the TSAPI Service error logs. Consult the logs for the return code. Call Customer Support and report this error. See Customer Support on page 21.
72	TSERVER_TDI_QUEUE_FAULT	TSAPI Service	This error indicates that there is a software problem with the TSAPI Service.	This error should never be returned to an application or appear in the TSAPI Service error logs. If this event is generated by the TSAPI Service, then there is a software problem with the TSAPI Service. Call Customer Support and report this error. See Customer Support on page 21.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
73	TSERVER_DRIVER_CONGESTION	TSAPI Service	The TSDI buffer is congested, which means that the amount of allocated TSDI space as reach the highwater mark. This occurs when the TSAPI Service is not processing messages fast enough.	1. Increase the TSDI space. In **CTI OAM > Administration > Security Database > Tlink > Edit Tlink**. 2. If the driver has indicated to the TSAPI Service that it can accept flow control information, you can change the default flow control level to a higher value. 3. If the driver still cannot handle the message flow, then check with your Customer Support for load capabilities of the TSAPI Service.
74	TSERVER_NO_TDI_BUFFERS	TSAPI Service	The TSAPI Service cannot allocate any more memory for the Tlink to which the application is connected. The driver registers an amount of memory with the TSAPI Service when it loads. The TSAPI Service uses this value as a maximum amount that can be allocated at one time.	1. Increase the TSDI space. In **CTI OAM > Administration > Security Database > Tlink > Edit Tlink**. 2. If the driver has indicated to the TSAPI Service that it can accept flow control information, you can change the default flow control level to a higher value. 3. If the driver can still not handle the message flow, call Customer Support.
75	TSERVER_OLD_INVOKEID	TSAPI Service	The TSAPI Service has received a message from a driver which contains an invokeID that it does not recognize. The TSAPI Service will still send this message to the application.	The TSAPI Service may be taking a very long time to respond to client requests. If this continues to happen call Customer Support.
76	TSERVER_HWMARK_TO_LARGE	TSAPI Service	The TSAPI Service attempted to set the high water mark for the TSDI size to a value that was larger than the TSDI size itself.	The TSAPI Service should have prevented the user from entering a TSDI size that was smaller than the high water mark. This error indicates a problem with the TSAPI Service itself.
77	TSERVER_SET_ECB_TO_LOW	TSAPI Service	Obsolete message.	Obsolete message.
78	TSERVER_NO_RECORD_IN_FILE	TSAPI Service	Obsolete message.	Obsolete message.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
79	TSERVER_ECB_OVERDUE	TSAPI Service	Obsolete message.	Obsolete message.
80	TSERVER_BAD_PW_ENCRYPTION	TSAPI Service	Obsolete message.	Obsolete message.
81	TSERVER_BAD_TSERV_PROTOCOL	TSAPI Service	A client application attempted to open a stream with a protocol version (apiVer field in acsOpenStream()) set to a value that the TSAPI Service does not support.	Use **OAM** > **Status and Control** > **Services Summary**. Select **TSAPI Service** > **Details**. From TSAPI Link Details page, select **TLink Status**. Check the Supported Protocols field on the Tlink Status page to see which protocol version the TSAPI Service supports. Compare this to the requirements of the client application.
82	TSERVER_TSERVER_BAD_DRIVER_PROTOCOL	TSAPI Service	A client application attempted to open a stream with a protocol version (apiVer field in acsOpenStream()) set to a value that the PBX Driver the stream was destined for does not support.	Use **OAM** > **Status and Control** > **Services Summary**. Select **TSAPI Service** > **Details**. From TSAPI Link Details page, select **TLink Status**. Check the Supported Protocols field on the Tlink Status page to see which protocol version the TSAPI Service supports. Compare this to the requirements of the client application.
83	TSERVER_TSERVER_BAD_TRANSPORT_TYPE	TSAPI Service	This indicates that the TSAPI Service is having a problem with the transport layer.	There is a serious system problem. These errors will appear in the TSAPI Service error logs. Call Customer Support and report this error. See Customer Support on page 21.
84	TSERVER_PDU_VERSION_MISMATCH	TSAPI Service	A client application attempted to use a TSAPI call that is not supported by the negotiated protocol version for the current connection.	Use **OAM** > **Status and Control** > **Services Summary**. Select **TSAPI Service** > **Details**. From TSAPI Link Details page, select **TLink Status**. Check the Supported Protocols field on the Tlink Status page to see which protocol version the TSAPI Service supports. Compare this to the requirements of the client application.
85	TSERVER_TSERVER_VERSION_MISMATCH	TSAPI Service	The application is sending a request which is not valid based on the TSAPI version negotiation performed when the stream was opened.	The application should verify that it is requesting the appropriate version of TSAPI and that the driver can support this version.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
86	TSERVER_LICENSE_MISMATCH	TSAPI Service	This is an internal error in the TSAPI Service.	These errors will appear in the TSAPI Service error logs. Call Customer Support and report this error. See Customer Support on page 21.
87	TSERVER_BAD_ATTRIBUTE_LIST	TSAPI Service	This is an internal error in the TSAPI Service.	These errors will appear in the TSAPI Service error logs. Report this error. See Customer Support on page 21.r.
88	TSERVER_BAD_TLIST_TYPE	TSAPI Service	This is an internal error in the TSAPI Service.	These errors will appear in the TSAPI Service error logs. Call Customer Support and report this error. See Customer Support on page 21.
89	TSERVER_BAD_PROTOCOL_FORMAT	TSAPI Service	A client application attempted to open a stream with a protocol version (apiVer field in acsOpenStream()) that was set to a format that the TSAPI Service could not decipher.	The application being used has a software problem.
90	TSERVER_OLD_TSLIB	TSAPI Service	A client application attempted to open a stream using an outdated version of the TSLIB software that is incompatible with the current TSLIB software.	Upgrade the client to the current version of the TSLIB.
91	TSERVER_BAD_LICENSE_FILE	TSAPI Service	Obsolete message.	Obsolete message.
92	TSERVER_NO_PATCHES	TSAPI Service	Obsolete message.	Obsolete message.
93	TSERVER_SYSTEM_ERROR	TSAPI Service	This indicates that the TSAPI Service has a software problem.	Call Customer Support and report this error. See Customer Support on page 21.
94	TSERVER_OAM_LIST_EMPTY	TSAPI Service	Obsolete message.	Obsolete message.
95	TSERVER_TCP_FAILED	TSAPI Service	The TSAPI Service has encountered an error with the TCP/IP transport.	These errors will appear in the TSAPI Service error logs. Call Customer Support and report this error. See Customer Support on page 21.
96	TSERVER_SPX_DISABLED	TSAPI Service	Obsolete message.	Obsolete message.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
97	TSERVER_TCP_DISABLED	TSAPI Service	Obsolete message.	Obsolete message.
98	TSERVER_REQUIRED_MODULES_NOT_LOADED	TSAPI Service	Obsolete message.	Obsolete message.
99	TSERVER_TRANSPORT_IN_USE_BY_OAM	TSAPI Service	Obsolete message.	Obsolete message.
100	TSERVER_NO_NDS_OAM_PERMISSION	TSAPI Service	Obsolete message.	Obsolete message.
101	TSERVER_OPEN_SDB_LOG_FAILED	TSAPI Service	Obsolete message.	Obsolete message.
102	TSERVER_INVALID_LOG_SIZE	TSAPI Service	Obsolete message.	Obsolete message.
103	TSERVER_WRITE_SDB_LOG_FAILED	TSAPI Service	Obsolete message.	Obsolete message.
104	TSERVER_NT_FAILURE	TSAPI Service	Obsolete message.	Obsolete message.
105	TSERVER_LOAD_LIB_FAILED	TSAPI Service	The TSAPI Service cannot load the G3PD.	Verify that the driver and its supporting DLLs are located in the system environment path. If the TSAPI Service software was just installed, try rebooting the server (Software Only). For a Bundled AE Server, contact Customer Support.
106	TSERVER_INVALID_DRIVER	TSAPI Service	Obsolete message.	Obsolete message.
107	TSERVER_REGISTRY_ERROR	TSAPI Service	Obsolete message.	Obsolete message.
108	TSERVER_DUPLICATE_ENTRY	TSAPI Service	Obsolete message.	Obsolete message.
109	TSERVER_DRIVER_LOADED	TSAPI Service	Obsolete message.	Obsolete message.
110	TSERVER_DRIVER_NOT_LOADED	TSAPI Service	Obsolete message.	Obsolete message.
111	TSERVER_NO_LOGON_PERMISSION	TSAPI Service	Obsolete message.	Obsolete message.
112	TSERVER_ACCOUNT_DISABLED	TSAPI Service	Obsolete message.	Obsolete message.
113	TSERVER_NO_NET_LOGON	TSAPI Service	Obsolete message.	Obsolete message.

Table 22: ACS Universal Failure Events (continued)

Error	Message	Type	Description	Corrective Action
114	TSERVER_ACCT_RESTRICTED	TSAPI Service	The account for accessing the TSAPI Service is restricted.	This may be due to too many failed login attempts. Make sure the user name and password are valid in your user authentication system (for example, the AE Services User Service or Active Directory Services).
115	TSERVER_INVALID_LOGON_TIME	TSAPI Service	Obsolete message.	Make sure the user name and password are valid in your user authentication system (for example, the AE Services User Service or Active Directory Services).Then wait and try to log in to the TSAPI Service at a later time.
116	TSERVER_INVALID_WORKSTATION	TSAPI Service	Obsolete message.	Obsolete message.
117	TSERVER_ACCT_LOCKED_OUT	TSAPI Service	The account has been locked out by the administrator.	Have the administrator reinstate the account, in your user authentication system (for example, the AE Services User Service or Active Directory Services).
118	TSERVER_PASSWORD_EXPIRED	TSAPI Service	The password has expired.	Change or update expiration information for the password in your user authentication system (for example, the AE Services User Service or Active Directory Services).
119	TSERVER_INVALID_HEARTBEAT_INTERVAL	TSAPI Service	The client has requested an invalid heartbeat interval.	Make sure you have set the value of the heartbeat interval (acsSetHeartbeatInterval()) to a valid value. The valid range of values is 5 to 60.

Table 23: ACS Related Errors

Error	Message	Type	Description
1000	DRIVER_DUPLICATE_ACSHANDLE	TSAPI Service	The acsHandle given for an ACS Stream request is already in use for a session. The already open session with the acsHandle is remains open.
1001	DRIVER_INVALID_ACS_REQUEST	TSAPI Service	The ACS message contains an invalid or unknown request. The request is rejected.
1002	DRIVER_ACS_HANDLE_REJECTION	TSAPI Service	The request is rejected because a CSTA request was issued with no prior acsOpenStream request or the acsHandle given for an acsOpenStream request is 0 or negative.
1003	DRIVER_INVALID_CLASS_REJECTION	TSAPI Service	The driver received a message containing an invalid or unknown message class. The request is rejected.
1004	DRIVER_GENERIC_REJECTION	TSAPI Service	The driver detected an invalid message for something other than message type or message class. This is an internal error and should be reported -- see Customer Support on page 21.
1005	DRIVER_RESOURCE_LIMITATION	TSAPI Service	The driver did not have adequate resources (that is memory, etc.) to complete the requested operation. This is an internal error and should be reported -- see Customer Support on page 21.
1006	DRIVER_ACSHANDLE_TERMINATION	TSAPI Service	Due to problems with the link to Communication Manager, the TSAPI Service has found it necessary to terminate the session with the given acsHandle. The session will be closed, and all outstanding requests will terminate.
1007	DRIVER_LINK_UNAVAILABLE	TSAPI Service	The TSAPI Service was unable to open the new session because no link was available to Communication Manager. The link may have been placed in the BLOCKED state, it may have been taken off line, or some other link failure may have occurred. When the link is in this state, the TSAPI Service remains loaded and advertised and sends this error for every new acsOpenStream request until the link becomes available again. A previously opened session will remain open when the link is in this state. It will receive no specific notification about the link status unless it attempts a CSTA request. In this state, a CSTA request will receive a CSTA Universal Failure with the error SERVICE_TERMINATION_REQUEST.

Appendix B: Summary of Private data support

This appendix provides historical information about private date versions in previous releases.

Private Data Version 7 features

AE Services TSAPI Service, Release 3.1, provides the following new features for Private Data Version 7.

- Network Call Redirection - see Network Call Redirection for Routing
- ISDN Redirecting Number - see Redirecting Number Information Element (presented through DeviceHistory)
- Query Device Name - see Query DeviceName for Attendants on page 813
- Enhanced Get API Capabilities function - see Enhanced GetAPICaps Version on page 813
- Expanded list of Auxiliary Work Reason codes - see Increased Aux Reason Codes on page 813

Network Call Redirection for Routing

The Adjunct Route support for Network Call Redirection capability allows an adjunct to request that an incoming trunk call be rerouted using the Network Call Redirection feature supported by the serving PSTN instead of having the call routed via a tandem trunk configuration. This support is provided by using the existing called party field with a new ASAI code point in the route-select message. For the list of TSAPI messages that this feature affects, see Private Data Version 7 features on page 814.

Redirecting Number Information Element (presented through DeviceHistory)

The “ISDN Redirecting Number for ASAI Events” Communication Manager feature will be used by CTI applications to provide enhanced treatment of incoming ISDN calls routed over an Integrated Services Digital Network (ISDN) facility. For the list of TSAPI messages that this feature affects, see Private Data Version 7 features on page 814.

To implement this feature, the TSAPI Service relies on a new parameter, called DeviceHistory. The TSAPI service uses the DeviceHistory parameter to provide the following information to applications:

- ISDN redirecting number
- the length of the device list
- merging rules

For more information about the DeviceHistory parameter, see Device History on page 146.

Query DeviceName for Attendants

The private Query DeviceName service allows an application to query the switch to identify the Integrated Directory name assigned to an extension.

With this version of private data, when a name has been assigned to an Attendant station extension, and an application issues a Query DeviceName service request. In the acknowledgement message, the deviceType parameter will contain DT_OTHER (a new value for PDV7) and the name parameter will contain the configured Integrated Directory name assigned to that attendant extension.

Enhanced GetAPICaps Version

The GetAPICaps function is enhanced to return the following information.

- Administered Switch Version
- Software Version
- Offer Type (values to be added in future releases of TSAPI Service)
- Server Type (more values to be added in future releases of TSAPI Service)

  This field will be a null string for DEFINITY systems. Valid values for Linux systems include: isp2100, premio, icc, laptop, ld380g3, hs20_8832_vm, hs20, ibmx305, ibmx306, and tn8400
- the maximum number of device history entries (deviceHistoryCount)

For the list of TSAPI messages that this feature affects, see Table 24: Private Data Version 7 features on page 814.

Increased Aux Reason Codes

AE Services supports the full range of Aux reason codes (values 0-99) that Communication Manager provides. Communication Manager returns a range of values from 0-99 in private data for the Query Agent State Confirmation Event and the Agent Logged Off event. Also, the private parameter “reasonCode” for the Set Agent State service request can be specified as a value from the wider range (0-99). The TSAPI Service will return whatever value is provided by the switch in a new private message. A new private message is required to the accommodate the new wider value (previously the range was 0-9). For the list of TSAPI messages that this feature affects, see Private Data Version 7 features on page 814.

Private Data Version 7 features and the updated services

Table 24 maps the Private Data Version 7 features to the services that they affect.

Table 24: Private Data Version 7 features

Private Data Version 7 feature	updated services
Network Call Redirection for Routing	• Conferenced Event on page 508 • Connection Cleared Event on page 528 • Delivered Event on page 536 • Diverted Event on page 572 • Established Event on page 584 • Failed Event on page 612 • Network Reached Event on page 630 • Queued Event on page 645 • Transferred Event on page 657 • Route Select Service (TSAPI Version 2) on page 733
ISDN Redirecting Number Information Element	• Conferenced Event on page 508 • Connection Cleared Event on page 528 • Delivered Event on page 536 • Diverted Event on page 572 • Established Event on page 584 • Failed Event on page 612 • Network Reached Event on page 630 • Queued Event on page 645 • Transferred Event on page 657 • Route Select Service (TSAPI Version 2) on page 733 • Snapshot Call Service on page 432
Query DeviceName for Attendants	• Query Device Name Service on page 399
Enhanced GetAPICaps Version	• CSTA Get API Capabilities confirmation structures for Private Data Version 7 on page 815
Increased Aux Reason Codes	• Logged Off Event on page 624 • Query Agent State Service on page 378 • Set Agent State Service on page 342

CSTA Get API Capabilities confirmation structures for Private Data Version 7

The TSAPI Service provides version-dependent private services in the CSTAGetAPICaps Confirmation private data interface. For Private Data Version 7 the GetAPICapsConfirmation Event has been updated to include the fields described in Table 25. See also, Code for the ATTGetAPICapsConfEvent - PDV 7 on page 816.

Table 25: New GetAPICapsConfirmation Event fields

New field	Description
`char adminSoftwareVersion[256];`	Administered switch software version. The value is 1 based, so the value passed will reflect the value that is administered. For example, if the switch version is administered to be 12, then 12 will be passed in the connection accepted message
`char softwareVersion[256];`	actual switch software version-- the same software version string that is shown when a customer logs into a SAT for a switch
`char offerType[256];`	Offer type. This field will be a null string for DEFINITY Servers. Valid values for Linux systems include: sray, seagull, chawk, chawk-lsp, s8500, s8500_blade, and vm_blade
`char serverType[256];`	Server type. This field will be a null string for DEFINITY Servers. Valid values for Linux systems include: isp2100, premio, icc, laptop, ld380g3, hs20_8832_vm, hs20, ibmx305, ibmx306, and tn8400
`unsigned char deviceHistoryCount`	Value of 1 to indicate how long the maximum length can be for a device history value. get deviceHistroy will be, at most, 1 long.

Code for the ATTGetAPICapsConfEvent - PDV 7

The ATT_Private_Identifiers.h file, which is provided in the AE Services TSAPI SDK contains the code for ATTGetAPICapsConfEvent. Here is the code for the ATTGetAPICapsConfEvent.

```
typedef struct ATTGetAPICapsConfEvent_t {
    char            switchVersion[65];
    unsigned char   sendDTMFTone;
    unsigned char   enteredDigitsEvent;
    unsigned char   queryDeviceName;
    unsigned char   queryAgentMeas;
    unsigned char   querySplitSkillMeas;
    unsigned char   queryTrunkGroupMeas;
    unsigned char   queryVdnMeas;
    unsigned char   singleStepConference;
    unsigned char   selectiveListeningHold;
    unsigned char   selectiveListeningRetrieve;
    unsigned char   setBillingRate;
    unsigned char   queryUCID;
    unsigned char   chargeAdviceEvent;
    unsigned char   singleStepTransfer
    unsigned char   reserved2;
    unsigned char   deviceHistoryCount;
    char            adminSoftwareVersion[256];
    char            softwareVersion[256];
    char            offerType[256];
    char            serverType[256];
} ATTGetAPICapsConfEvent_t;
```

Private Data Version Feature Support prior to AE Services TSAPI R3.1.0

All currently supported Communication Manager servers provide call prompting digits, the only private data item in version 1. Private data versions 2 through 6 encompass a much broader feature set, where some features may be dependent upon the switch version.

- Private data version 2 includes support for some features that are available only on the G3V3 and later releases.
- Private data versions 3 and 4 include support for some features that are available only with the G3V4 and later releases.
- Private data version 5 includes support for some features that are available only on the G3V5, G3V6, G3V7 and later releases.
- Private data version 6 includes support for some features that are available only on the G3V8 and later releases.

Summary of private data versions 2 through 6

Table 26 provides a complete list of private data features prior to Application Enablement Services (AE Services) 3.1.0. The associated initial DEFINITY (or Communication Manager) and G3PD releases that support each one are included, as well as the version of private data in which the feature was first introduced.

Table 26: Private Data Summary

Private Data Feature	Initial DEFINITY or Communication Manager Release	Initial DEFINITY PBX Driver Release	Initial Private Data Version
Prompted Digits in Delivered Events	All	R2.1 (private data)	V1
Priority, Direct Agent, Supervisor Assist Calling	All	R2.1 (private data)	V2
Enhanced Call Classification	All	R2.1 (private data)	V2
Trunk, Classifier Queries	All	R2.1 (private data)	V2
LAI in Events	All	R2.1 (private data)	V2
Launching Predictive Calls from Split	All	R2.1 (private data)	V2
Application Integration with Expert Agent Selection	G3V3	R2.1 (private data)	V2
User-to-User Info (Reporting and Sending)	G3V3	R2.1 (private data)	V2
Multiple Notification Monitors (two on ACD/VDN)	G3V3	All	V2
Launching Predictive Calls from VDN	G3V3	R2.1	V2
Multiple Outstanding Route Requests for One Call	G3V3	R2.1	V2
Answering Machine Detection	G3V3	R2.1 (private data)	V2
Established Event for Non-ISDN Trunks	G3V3	All	V2
Provided Prompter Digits on Route Select	G3V3	R2.1 (private data)	V2
Requested Digit Selection	G3V3	R2.1 (private data)	V2

Table 26: Private Data Summary (continued)

Private Data Feature	Initial DEFINITY or Communication Manager Release	Initial DEFINITY PBX Driver Release	Initial Private Data Version
VDN Return Destination (Serial Calling)	G3V3	R2.1 (private data)	V2
Deflect Call	G3V4	R2.2	V3
Pickup Call	G3V4	R2.2	V3
Originated Event Report	G3V4	R2.2	V3
Agent Logon Event Report	G3V4	R2.2 (private data)	V3
Reason for Redirection in Alerting Event Report	G3V4	R2.2 (private data)	V3
Agent, Split, Trunk, VDN Measurements Query	G3V4	R2.2 (private data)	V3
Device Name Query	G3V4	R2.2 (private data)	V3
Send DTMF Tone	G3V4	R2.2 (private data)	V3
Distributing Device in Conferenced, Delivered, Established, and Transferred Events	All	R2.2 (private data)	V4
G3 Private Capabilities in cstaGetAPICaps Confirmation Private Data	G3V3	R2.2 (private data)	V4
Support Detailed DeviceIDType_t in Events	G3V3	R3.10 (private data)	V5
Set Bill Rate	G3V4	R3.10 (private data)	V5
Flexible Billing in Delivered Event, Established Event, and Route Request	G3V4	R3.10 (private data)	V5
Call Originator Type in Delivered Event, Established Event, and Route Request	G3V4	R3.10 (private data)	V5
Selective Listening Hold	G3V5	R3.10 (private data)	V5
Selective Listening Retrieve	G3V5	R3.10 (private data)	V5
Set Advice of Charge	G3V5	R3.10 (private data)	V5
Charge Advice Event	G3V5	R3.10 (private data)	V5

Table 26: Private Data Summary (continued)

Private Data Feature	Initial DEFINITY or Communication Manager Release	Initial DEFINITY PBX Driver Release	Initial Private Data Version
Reason Code in Set Agent State, Query Agent State, and Logout Event	G3V5	R3.10 (private data)	V5
27-Character Display Query Device Name Confirmation	G3V5	R3.10 (private data)	V5
Unicode Device ID in Events	G3V6	R3.10 (private data)	V5
Trunk Group and Trunk Member Information in Network Reached Event	G3V6	R3.10 (private data)	V5
Universal Call ID (UCID) in Events	G3V6	R3.10 (private data)	V5
Single Step Conference	G3V6	R3.10 (private data)	V5
Pending Work Mode and Pending Reason Code in Set Agent State and Query Agent State	G3V8	R3.30 (private data)	V6
Trunk Group and Trunk Member Information in Delivered Event and Established Event regardless of whether Calling Party is Available	G3V8	R3.30 (private data)	V6
Trunk Group Information in Route Request Events regardless of whether Calling Party is Available	G3V8	R3.30 (private data)	V6
Trunk Group Information for Every Party in Transferred Events and Conferenced Events	G3V8	R3.30 (private data)	V6
User-to-User Info (UUI) is increased from 32 to 96 bytes	G3V8	R3.30 (private data)	V6

Table 27: Renaming PDUs and structures - Private Data Version 7

1 If your code contains these PDUs and structure member names	2 Rename them as follows:
ATT_CONFERENCED ATTConferencedEvent_t conferencedEvent	ATTV6_CONFERENCED ATTV6ConferencedEvent_t v6conferencedEvent
ATT_CONNECTION_CLEARED ATTConnectionClearedEvent_t connectionClearedEvent	ATTV6_CONNECTION_CLEARED ATTV6ConnectionClearedEvent_t v6connectionclearedEvent
ATT_DELIVERED ATTDeliveredEvent_t deliveredEvent	ATTV6_DELIVERED ATTV6DeliveredEvent_t v6deliveredEvent
ATT_ESTABLISHED ATTEstablishedEvent _t establishedEvent	ATTV6_ESTABLISHED ATTV6EstablishedEvent_t v6establishedEvent
ATT_NETWORK_REACHED ATTNetworkReachedEvent_t networkReached	ATTV6_NETWORK_REACHED ATTV6NetworkReachedEvent_t v6networkReached
ATT_TRANSFERRED ATTTransferredEvent_t transferredEvent	ATTV6_TRANSFERRED ATTV6TransferredEvent _t v6transferredEvent
ATT_ROUTE_REQUEST ATTRouteRequestEvent routeSelectReq	ATTV6_ROUTE_REQUEST ATTV6RouteRequestEvent v6routeSelectReq
ATT_QUERY_DEVICE_NAME_CONF ATTQueryDeviceNameConfEvent_t queryDeviceName	ATTV6_QUERY_DEVICE_NAME_CONF ATTV6QueryDeviceNameConfEvent_t v6queryDeviceName
ATT_GETAPI_CAPS_CONF ATTGetAPICapsConfEvent_t getAPICaps	ATTV6_GETAPI_CAPS_CONF ATTV6GetAPICapsConfEvent_t v6getAPICaps

Table 28: Migration of PDV 5 PDUs and Structure Members to PDV 6

Original V5 PDU or Structure Member Name	Required Changes to V5 PDU or Structure Member Name for V6 Interface	New V6 PDU or Structure Member Name
ATT_QUERY_AGENT_STATE_CONF ATTQueryAgentStateConfEvent_t queryAgentState	ATTV5_QUERY_AGENT_STATE_CONF ATTV5QueryAgentStateConfEvent_t v5queryAgentState	ATT_QUERY_AGENT_STATE_CONF ATTQueryAgentStateConfEvent_t queryAgentState
ATT_SET_AGENT_STATE ATTSetAgentState_t setAgentStateReq	ATTV5_SET_AGENT_STATE ATTV5SetAgentState_t v5setAgentStateReq	ATT_SET_AGENT_STATE ATTSetAgentState_t setAgentStateReq
N/A	New for V6	ATT_SET_AGENT_STATE_CONF ATTSetAgentStateConfEvent_t
ATT_ROUTE_REQUEST ATTRouteRequestEvent_t	ATTV5_ROUTE_REQUEST ATTV5RouteRequestEvent_t	ATT_ROUTE_REQUEST ATTRouteRequestEvent_t
ATT_TRANSFERRED ATTTransferredEvent_t	ATTV5_TRANSFERRED ATTV5TransferredEvent_t	ATT_TRANSFERRED ATTTransferredEvent_t
ATT_CONFERENCED ATTConferencedEvent_t	ATTV5_CONFERENCED ATTV5ConferencedEvent_t	ATT_CONFERENCED ATTConferencedEvent_t
ATT_CLEAR_CONNECTION ATTClearConnection_t	ATTV5_CLEAR_CONNECTION ATTV5ClearConnection_t	ATT_CLEAR_CONNECTION ATTClearConnection_t
ATT_CONSULTATION_CALL ATTConsultationCall_t	ATTV5_CONSULTATION_CALL ATTConsultationCall_t	ATT_CONSULTATION_CALL ATTConsultationCall_t
ATT_MAKE_CALL ATTMakeCall_t	ATTV5_MAKE_CALL ATTV5MakeCall_t	ATT_MAKE_CALL ATTMakeCall_t
ATT_DIRECT_AGENT_CALL ATTDirectAgentCall_t	ATTV5_DIRECT_AGENT_CALL ATTV5DirectAgentCall_t	ATT_DIRECT_AGENT_CALL ATTDirectAgentCall_t
ATT_MAKE_PREDICTIVE_CALL ATTMakePredictiveCall_t	ATTV5_MAKE_PREDICTIVE_CALL ATTV5MakePredictiveCall_t	ATT_MAKE_PREDICTIVE_CALL ATTMakePredictiveCall_t
ATT_SUPERVISOR_ASSIST_CALL ATTSupervisorAssistCall_t	ATTV5_SUPERVISOR_ASSIST_CALL ATTV5SupervisorAssistCall_t	ATT_SUPERVISOR_ASSIST_CALL ATTSupervisorAssistCall_t
ATT_RECONNECT_CALL ATTReconnectCall_t	ATTV5_RECONNECT_CALL ATTV5ReconnectCall_t	ATT_RECONNECT_CALL ATTReconnectCall_t
ATT_CONNECTION_CLEARED ATTConnectionClearedEvent_t	ATTV5_CONNECTION_CLEARED ATTV5ConnectionClearedEvent_t	ATT_CONNECTION_CLEARED ATTConnectionClearedEvent_t
ATT_ROUTE_SELECT ATTRouteSelect_t	ATTV5_ROUTE_SELECT ATTV5RouteSelect_t	ATT_ROUTE_SELECT ATTRouteSelect_t
ATT_DELIVERED ATTDeliveredEvent_t	ATTV5_DELIVERED ATTV5DeliveredEvent_t	ATT_DELIVERED ATTDeliveredEvent_t

Table 28: Migration of PDV 5 PDUs and Structure Members to PDV 6 (continued)

Original V5 PDU or Structure Member Name	Required Changes to V5 PDU or Structure Member Name for V6 Interface	New V6 PDU or Structure Member Name
ATT_ESTABLISHED ATTEstablishedEvent_t	ATTV5_ESTABLISHED ATTV5EstablishedEvent_t	ATT_ESTABLISHED ATTEstablishedEvent_t
ATT_ORIGINATED ATTOriginatedEvent_t	ATTV5_ORIGINATED ATTV5OriginatedEvent_t	ATT_ORIGINATED ATTOriginatedEvent_t

CSTA Device ID Type (Private Data Version 4 and Earlier)

If an application opens an ACS stream with Private Data version 4 and earlier, G3PD supports only a limited number of types of DeviceIDType_t for the deviceIDType parameter of an ExtendedDeviceID_t. The types supported are described in Table 29.

Table 29: CSTA Device Type and Status (Private Data Version 4 and Earlier)

DeviceIDType_t	ConnectionID_Device_t	DeviceIDStatus_t	Type of Devices
DEVICE_IDENTIFIER	STATIC_ID	ID_PROVIDED	Internal or external endpoints that have a known device identifier
TRUNK_IDENTIFIER	DYNAMIC_ID	ID_PROVIDED	Internal or external endpoints that do not have a known device identifier
EXPLICIT_PUBLIC_UNKNOWN		ID_NOT_KNOWN or ID_NOT_REQUIRED	

CSTAGetAPICaps Confirmation interface structures for Private Data Versions 4, 5, and 6

Beginning with private data version 4, the TSAPI Service provides the Communication Manager version-dependent private services in the CSTAGetAPICaps Confirmation private data interface, as defined by the following structures:

Private Data Version 5 and 6 Syntax on page 825

Private Data Version 4 Syntax on page 826

Private Data Version 5 and 6 Syntax

```
typedef struct ATTGetAPICapsConfEvent_t
{
char switchVersion[16];// specifies the switch
                              // version - G3V2, G3V3,
                              // G3V3, G3V4, G3V5,
                              // G3V6 or G3V8. (no new
                              // capabilities are pro-
                              // vided with G3V7 so the
                              // G3 PBX driver does not
                              // differentiate between
                              // a G3V6 and a G3V7.
Boolean sendDTMFTone;// TRUE - supported,
                              // FALSE - not supported
Boolean enteredDigitsEvent;// TRUE - supported,
                                    // FALSE - not supported
Boolean queryDeviceName;            // TRUE - supported,
                                    // FALSE - not supported
Boolean queryAgentMeas;             // TRUE - supported,
                                    // FALSE - not supported
Boolean querySplitSkillMeas; // TRUE - supported,
                                    // FALSE - not supported
Boolean queryTrunkGroupMeas; // TRUE - supported,
                                    // FALSE - not supported
Boolean queryVdnMeas;               // TRUE - supported,
                                    // FALSE - not supported
Boolean singleStepConference;// TRUE - supported,
                                    // FALSE - not supported
Boolean selectiveListeningHold;// TRUE - supported,
                                    // FALSE - not supported
Boolean selectiveListeningRetrieve;// TRUE - supported
                                    // FALSE - not supported
Boolean setBillingRate;             // TRUE - supported,
                                    // FALSE - not supported
Boolean queryUcid;                  // TRUE - supported,
                                    // FALSE - not supported
Boolean chargeAdviceEvent;// TRUE - supported,
                                    // FALSE - not supported
Boolean reserved1;                  // reserved for future use
Boolean reserved2;                  // reserved for future use
} ATTGetAPICapsConfEvent_t;
```

Private Data Version 4 Syntax

```
typedef struct ATTV4GetAPICapsConfEvent_t
{
     char switchVersion[16];                    // specifies the switch
                                                // version - G3V2, G3V3,
                                                // G3V3, G3V4, G3V5, or
                                                // G3V6
     Boolean sendDTMFTone;                      // TRUE - supported,
                                                // FALSE - not supported
     Boolean enteredDigitsEvent;                // TRUE - supported,
                                                // FALSE - not supported
     Boolean queryDeviceName;                   // TRUE - supported,
                                                // FALSE - not supported
     Boolean queryAgentMeas;                    // TRUE - supported,
                                                // FALSE - not supported
     Boolean querySplitSkillMeas;               // TRUE - supported,
                                                // FALSE - not supported
     Boolean queryTrunkGroupMeas;               // TRUE - supported,
                                                // FALSE - not supported
     Boolean queryVdnMeas;                      // TRUE - supported,
                                                // FALSE - not supported
     Boolean reserved1;                         // reserved for future use
     Boolean reserved2;                         // reserved for future use
} ATTV4GetAPICapsConfEvent_t;
```

Note:

Communication Manager capabilities are obtained only once when the G3PD is loaded during negotiation with the switch. If the G3PD is not unloaded and reloaded after the switch software version is changed (for example, from G3V3 to G3V4 or vice versa), then once this change is made, and the G3PD is not unloaded and reloaded again, the cstaGetAPICaps requests will return the capabilities that the G3PD obtained when it was first loaded. Thus cstaGetAPICaps will not reflect the real capabilities of the new switch version.

Private Data Function Changes between V5 and V6

Please note that the following Private Data functions are changed between V5 and V6.

Set Agent State

```
// attSetAgentState() - Private Data V5 Interface

RetCode_t    attSetAgentStateExt(// old function name used in V5 API
    ATTPrivateData_t*attPrivateData,
    ATTWorkMode_t   workMode,
    long            reasonCode); // new parameter in V5 API

// attSetAgentStateExt() - Private Data V6 Interface

RetCode_t    attV6SetAgentState(// new function name used in V6 API
    ATTPrivateData_t*attPrivateData,
    ATTWorkMode_t   workMode,
    long            reasonCode,
    Boolean         enablePending);// new parameter in V6 API
```

Private Data Sample Code

This section provides the following examples of Private Data sample code:

Sample Code 1

```
#include <stdio.h>

#include <acs.h>
#include <csta.h>
#include <attpriv.h>

/*
* Make Direct Agent Call  - from "12345" to  ACD Agent extension "11111"
*           - ACD agent must be logged into split "22222"
*           - no User to User info
*           - not a priority call
*/

ACSHandle_t acsHandle;              // An opened ACS Stream Handle
InvokeID_t  invokeID = 1;           // Application generated
                                    // Invoke ID
DeviceID_t  calling = "12345";      // Call originator, an on-PBX
                                    // extension
DeviceID_t  called  = "11111";      // Call destination, an ACD
                                    // Agent extension
DeviceID_t  split = "22222";        // ACD Agent is logged into
                                    // this split
Boolean     priorityCall = FALSE;   // Not a priority call
RetCode_t   retcode;                // Return code for service
                                    // requests
CSTAEvent_t cstaEvent;              // CSTA event buffer
unsigned    short       eventBufSize; // CSTA event buffer size
ATTPrivateData_tprivateData;        // ATT service request private
                                    // data buffer

            retcode = attDirectAgentCall(&privateData, &split, priorityCall,
                        NULL);

            if ( retcode < 0 ) {
            /* Some kind of failure, need to handle this ... */
            }
```

```
retcode = cstaMakeCall(acsHandle, invokeID, &calling, &called,
```

Sample Code 1 (continued)

```
(PrivateData_t *)&privateData);

            if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}
/*  Make Call request succeeded.  Wait for confirmation event. */

eventBufSize = sizeof(CSTAEvent_t);
privateData.length = ATT_MAX_PRIVATE_DATA;

retcode = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
            &eventBufSize, (PrivateData_t *)&privateData, NULL);

if (retcode != ACSPOSITIVE_ACK) {
/* Some kind of failure, need to handle this ... */
}

if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
            (cstaEvent.eventHeader.eventType == CSTA_MAKE_CALL_CONF)) {
              if (cstaEvent.event.cstaConfirmation.invokeID == 1) {
            /*
             * Invoke ID matches, Make Call is confirmed.
            /*
                    }
            }
```

Sample Code 2

```
#include <stdio.h>

#include <acs.h>
#include <csta.h>
#include <attpriv.h>

/*
 *  Set Agent State - Request to log in ACD Agent with initial work mode
 *  "Auto-In".
 */

ACSHandle_t acsHandle;              // An opened ACS Stream Handle
InvokeID_t  invokeID = 1;           // Application generated
                                    // Invoke ID
DeviceID_t  device = "12345";       // Device associated with
                                    // ACD Agent
AgentMode_t agentMode = AM_LOG_IN;  // Requested Agent Mode
AgentID_t   agentID = "01";         // Agent login identifier
AgentGroup_tagentGroup = "11111";   // ACD split to log Agent into
AgentPassword_t *agentPassword = NULL;// No password, i.e., not EAS
RetCode_t   retcode;                // Return Code for service
                                    // requests
CSTAEvent_t cstaEvent;              // CSTA event buffer
unsigned shorteventBufSize;         // CSTA event buffer size
ATTPrivateData_tprivateData;        // ATT service request private
// data buffer

retcode = attV6SetAgentState(&privateData, WM_AUTO_IN, 0, TRUE);

if (retcode < 0 ) {
            /* Some kind of failure, need to handle this ... */
            }

retcode = cstaSetAgentState(acsHandle, invokeID, &device, agentMode,
            &agentID, &agentGroup, agentPassword,
            (PrivateData_t *)&privateData);

if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}

}
```

Sample Code 2 (Continued)

```
/* Set Agent State request succeeded.  Wait for confirmation event.*/

eventBufSize = sizeof(CSTAEvent_t);
privateData.length = ATT_MAX_PRIVATE_DATA;

retcode = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
            &eventBufSize, (PrivateData_t *)&privateData, NULL);

if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}

if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
            (cstaEvent.eventHeader.eventType == CSTA_SET_AGENT_STATE_CONF)) {
             if (cstaEvent.event.cstaConfirmation.invokeID == 1) {
            /*
            * Invoke ID matches, Set Agent State is confirmed.
            * Private data is returned in confirmation event.
            */
            if (privateData.length > 0) {
            /* Confirmation contains private data */

            if (attPrivateData(&privateData, &attEvent) != ACSPOSITIVE_ACK) {
            /* Decoding error */
            }
            else { // See whether the requested change is pending or not
            ATTSetAgentStateConfEvent_t *setAgentStateConf ;
            SetAgentStateConf = &privateData.u.setAgentState;
            if (SetAgentStateConf->isPending == TRUE)
            // The request is pending
}
                    }
                        }
```

Sample Code 3

```
#include <stdio.h>

#include <acs.h>
#include <csta.h>
#include <attpriv.h>

/*
  * Query ACD Split via cstaEscapeService()
 */

ACSHandle_t acsHandle;              // An opened ACS Stream Handle
InvokeID_t  invokeID = 1;           // Application generated
                                    // Invoke ID
DeviceID_t  deviceID = "12345";     // Device associated with
                                    // ACD split
RetCode_t   retcode;                // Return code for service
                                    // requests
CSTAEvent_t cstaEvent;              // CSTA event buffer
unsigned shorteventBufSize;         // CSTA event buffer size
ATTPrivateData_tprivatedata;        // ATT private data buffer
ATTEvent_t  attEvent;               // ATT event buffer
ATTQueryAcdSplitConfEvent_t         // Query ACD Split confirmation
            *queryAcdSplitConf;     // event pointer


retcode = attQueryAcdSplit(&privatedata, &deviceID);

if (retcode < 0 ) {
            /* Some kind of failure, need to handle this ... */
}

retcode = cstaEscapeService(acsHandle, invokeID,
            (PrivateData_t *)&privateData);

if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}
```

Sample Code 3 (Continued)

```
/*
* Now wait for confirmation event.
*
* To retrieve private data return parameters for Query ACD Split,
* the application must specify a pointer to a private data buffer as
* a parameter to either the acsGetEventBlock() or acsGetEventPoll()
* request. Upon return, the application passes the address
* to attPrivateData() for decoding.
*/

eventBufSize = sizeof(CSTAEvent_t);
privateData.length = ATT_MAX_PRIVATE_DATA;

retcode = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
            &eventBufSize, (PrivateData_t *)&privateData, NULL);

if (retcode != ACSPOSITIVE_ACK) {
            /* Some kind of failure, need to handle this ... */
}

if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
            (cstaEvent.eventHeader.eventType == CSTA_ESCAPE_SVC_CONF)) {
              if (cstaEvent.event.cstaConfirmation.invokeID != 1) {
            /* Error - wrong invoke ID */
            }
            else if (privateData.length > 0) {
            /* Confirmation contains private data */

if (attPrivateData(&privateData, &attEvent) != ACSPOSITIVE_ACK) {
            /* Decoding error */
            }
            else if (attEvent.eventType == ATT_QUERY_ACD_SPLIT_CONF) {
            queryAcdSplitConf = (ATTQueryAcdSplitConfEvent_t *)
            &attEvent.u.queryAcdSplit;
            }
            }
            else {
            /* Error - no private data in confirmation event */
            }
}
```

Appendix C: Server-Side Capacities

This appendix describes server-side capacities, which include Avaya Communication Manager capacities and AE Services TSAPI Service capacities.

Communication Manager CSTA system capacities

Table 30 provides Communication Manager CSTA System Capacities. These are maximum system capacities. The numbers shown, as well as the server's hardware configuration and the switch configuration, limit a telephony server's capacity. The number of users that can access a telephony server is independent of these numbers. User access to the TSAPI Service may be limited by the AE Services purchase agreement.

Table 30: Communication CSTA System Capacities

Parameter	S8000 series Servers	DEFINITY Server G3i System Capacity	DEFINITY Server System Capacity	DEFINITY Server G3r System Capacity	TSAPI Service	Comments
CTI Links	64	Eight	Four	Eight	16	Up to 16 switch connections are supported by the TSAPI Service.
CSTA Service Requests per CTI Link		Limited by the following numbers	Limited by the following numbers	Limited by the following numbers	2000 per CTI link	See Note 1 on page 837
Objects monitored by cstaMonitorDevice requests		2000 per DEFINITY Server	250 per DEFINITY Server	6000 per DEFINITY Server	Limited by the lesser of switch capacity and link capacity	Maximum number of monitored stations. See Note 2 on page 838
Objects monitored by cstaMonitorCallsViaDevice requests		170 per DEFINITY Server	50 per DEFINITY Server	• 460 per DEFINITY Server • 2000 for release G3V3 and later	Limited by switch capacity and link capacity	Maximum number of monitored VDNs and ACD splits allowed. See Note 2.
Simultaneous cstaMonitorDevice monitor requests on one station device		Two, but the TSAPI Service multiplexes client requests into a single association.	Two, but a TSAPI Service multiplexes client requests into a single association.	Two, but the TSAPI Service multiplexes client requests into a single association.	No maximum number but limited by TSAPI Service memory.	Monitor requests can come from the same application or different applications.
Simultaneous cstaMonitorCallsViaDevice monitor requests on one ACD device		One, but the TSAPI Service multiplexes client requests into a single association.	One, but the TSAPI Service multiplexes client requests into a single association.	One, but the TSAPI Service multiplexes client requests into a single association.	No maximum number but limited by TSAPI Service memory	Monitor requests can come from the same application or different applications.

Table 30: Communication CSTA System Capacities (continued)

Parameter	S8000 series Servers	DEFINITY Server G3i System Capacity	DEFINITY Server System Capacity	DEFINITY Server G3r System Capacity	TSAPI Service	Comments
Simultaneous CSTA Clear Connection, Clear Call, and Set Feature Service requests		300 per G3 switch	75 per G3 switch	3000 per G3 switch	Limited by switch capacity and link capacity	See Note 3 on page 838.
Simultaneous CSTA service requests other than the ones listed in the preceding table cell		2000 per G3 switch	250 per G3 switch	6000 per G3 switch	Limited by the lesser of switch capacity and link capacity	Maximum number of monitored stations See Note 2.
Number of simultaneous call classifications in progress (predictive calls in between the make call request and the switch returning a classification)		40 per G3 switch	40 per G3 switch	400 per G3 switch	N/A	
Number of simultaneous outstanding route requests on a G3 CTI link		127	127	4000 (G3V8 and later)	127 (G3V7 and earlier)	
Number of devices that can be on a call		Six	Six	Six	Six	See Note 4 on page 838.
Number of cstaMonitor CallsViaDevice monitored objects that can be involved in a call		One per G3 switch; three for G3V3 or later	One per G3 switch; three for G3V3 or later	One per G3 switch; three for G3V3 or later	The TSAPI Service multiplexes client requests into a single association.	A call can only be actively monitored via one ACD device. See Note 5 on page 838.
Number of CSTA monitor requests that can be involved in a call		N/A	N/A	N/A	No maximum number but limited by TSAPI Service memory	Each CSTA Event Report of a monitored object will be sent to every monitor request.

Note 1

This number (CSTA Service Requests per CTI Link) consists of all Monitored objects, outstanding Call Control Service requests, and outstanding Set Feature Service requests (as well as the outstanding requests of Query Services and Routing Service). The default number set for each CTI link is 2000. This number can be higher or lower depending on the memory configuration of the TSAPI Service. This number should be configured according to the administration information in <*document name*>.

Note 2

This is not the number of total monitor requests. An object monitored by multiple monitor requests is counted only once. All Call Control Service requests on a station device other than Clear Connection and Clear Call are included in this number. When a station device is monitored, the Call Control Service requests on the device are not counted as additional requests.

Note 3

This is an estimated number. This number includes all outstanding Clear Connection Service requests, Set Feature Service requests, and Query Service requests.

Note 4

A call can have a maximum of six parties.

Note 5

A call may pass through several ACD devices monitored by cstaMonitorCallsViaDevice requests, but only one is active (that is, receives event reports for that call) for that call at one time.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W