



Avaya Aura[®] Application Enablement Services

TSAPI for Avaya Communication Manager Programmer's Reference

Issue 4
March 2021

© 2020 Avaya Inc.
All Rights Reserved.

Notice

While reasonable efforts have been made to ensure that the information in this document is complete and accurate at the time of printing, Avaya assumes no liability for any errors. Avaya reserves the right to make changes and corrections to the information in this document without the obligation to notify any person or organization of such changes.

Documentation disclaimer

Avaya shall not be 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. 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 End User.

Link disclaimer

Avaya is not responsible for the contents or reliability of any linked Web sites referenced within this site or documentation(s) provided by Avaya. Avaya is not responsible for the accuracy of any information, statement or content provided on these sites and does not necessarily endorse the products, services, or information described or offered within them.

Avaya does not guarantee that these links will work all the time and has no control over the availability of the linked pages.

Warranty

Avaya 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 to Avaya customers and other parties through the Avaya Support Web site:

<http://www.avaya.com/support>. Please note that if you acquired the product from an authorized Avaya reseller outside of the United States and Canada, the warranty is provided to you by said Avaya reseller and not by Avaya.

Licenses

THE SOFTWARE LICENSE TERMS AVAILABLE ON THE AVAYA WEBSITE, [HTTP://SUPPORT.AVAYA.COM/LICENSEINFO/](http://support.avaya.com/licenseinfo/) ARE APPLICABLE TO ANYONE WHO DOWNLOADS, USES AND/OR INSTALLS AVAYA SOFTWARE, PURCHASED FROM AVAYA INC., ANY AVAYA AFFILIATE, OR AN AUTHORIZED AVAYA RESELLER (AS APPLICABLE) UNDER A COMMERCIAL AGREEMENT WITH AVAYA OR AN AUTHORIZED AVAYA RESELLER. UNLESS OTHERWISE AGREED TO BY AVAYA IN WRITING, AVAYA DOES NOT EXTEND THIS LICENSE IF THE SOFTWARE WAS OBTAINED FROM ANYONE OTHER THAN AVAYA, AN AVAYA AFFILIATE OR AN AVAYA AUTHORIZED RESELLER, AND AVAYA RESERVES THE RIGHT TO TAKE LEGAL ACTION AGAINST YOU AND ANYONE ELSE USING OR SELLING THE SOFTWARE WITHOUT A LICENSE. BY INSTALLING, DOWNLOADING OR USING THE SOFTWARE, OR AUTHORIZING OTHERS TO DO SO, YOU, ON BEHALF OF YOURSELF AND THE ENTITY FOR WHOM YOU ARE INSTALLING, DOWNLOADING OR USING THE SOFTWARE (HEREINAFTER REFERRED TO INTERCHANGEABLY AS "YOU" AND "END USER"), AGREE TO THESE TERMS AND CONDITIONS AND CREATE A BINDING CONTRACT BETWEEN YOU AND AVAYA INC. OR THE APPLICABLE AVAYA AFFILIATE ("AVAYA").

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 originally sold by Avaya and ultimately utilized by End User.

License type(s)

Named User License (NU). End User may: (i) install and use the Software on a single Designated Processor or Server per authorized Named User (defined below); or (ii) install and use the Software on a Server so long as only authorized Named Users access and use the Software. "Named User," means a user or device that has been expressly authorized by Avaya to access and use the Software. At Avaya's sole discretion, a "Named User" may be, without limitation, designated by name, corporate function (e.g., webmaster or helpdesk), an e-mail or voice mail account in the name of a person or corporate function, or a directory entry in the administrative database utilized by the Software that permits one user to interface with the Software.

Shrinkwrap License (SR). With respect to Software that contains elements provided by third party suppliers, End User may install and use the Software in accordance with the terms and conditions of the applicable license agreements, such as "shrinkwrap" or "clickwrap" license accompanying or applicable to the Software ("Shrinkwrap License"). The text of the Shrinkwrap License will be available from Avaya upon End User's request (see "Third-party Components" for more information).

Copyright

Except where expressly stated otherwise, no use should be made of materials on this site, the Documentation(s) and Product(s) provided by Avaya. All content on this site, the documentation(s) and the product(s) provided by Avaya including the selection, arrangement and design of the content is owned either by Avaya or its licensors and is protected by copyright and other intellectual property laws including the sui generis rights relating to the protection of databases. You may not modify, copy, reproduce, republish, upload, post, transmit or distribute in any way any content, in whole or in part, including any code and software. Unauthorized reproduction, transmission, dissemination, storage, and or use without the express written consent of Avaya 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 regarding distributed Linux OS source code (for those Products that have distributed the Linux OS source code), and identifying the copyright holders of the Third Party Components and the Third Party Terms that apply to them is available on the Avaya Support

Web site: <http://www.avaya.com/support/Copyright/>.

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/>. Suspected security vulnerabilities with Avaya products should be reported to Avaya by sending mail to: securityalerts@avaya.com.

Trademarks

Avaya, the Avaya logo, one-X Portal, Communication Manager, Application Enablement Services, Modular Messaging, and Conferencing are either registered trademarks or trademarks of Avaya Inc. in the United States of America and/or other jurisdictions.

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>

Contact 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 Web site: <http://www.avaya.com/support>

Contents

About this document	1
Intended audience.....	1
Structure and organization of this document	2
What's New for the AE Services Release 8.1.11 TSAPI Service	4
What's New for the AE Services Release 8.1.3 TSAPI Service	4
What Was New in Earlier Releases of the TSAPI Service	4
What's New for the AE Services Release 8.1.2 TSAPI Service	4
What's New for the AE Services Release 8.1 TSAPI Service	4
What's New for the AE Services Release 7.1.1 TSAPI Service	5
What's New for the AE Services Release 7.0 TSAPI Service	5
What's New for the AE Services Release 6.3.3 TSAPI Service	6
What's New for the AE Services Release 6.3.1 TSAPI Service	8
What's New for the AE Services Release 6.2.0 TSAPI Service	9
What's New for the AE Services Release 6.1.0 TSAPI Service	9
What's New for the AE Services Release 5.2.0 TSAPI Service	10
What's New for the AE Services Release 4.1.0 TSAPI Service	10
What's New for the AE Services 4. 0 TSAPI Service	12
AE Services 8.1.3 clients and backward compatibility	13
About installing the SDK	13
Related Documents.....	14
Related Ecma International documents	14
Related Avaya documents	15
Web based training	16
Customer Support	16
Conventions used in this document	17
Format of Service Description Pages	18
Common ACS Parameter Syntax	21
Chapter 1: Overview of the TSAPI Client and the TSAPI SDK	23
Introduction.....	23
Ecma International and the CSTA Standards	24
The TSAPI Specification	24
TSAPI for Avaya Communication Manager	24
The TSAPI Client	25
The TSAPI SDK	25
64 bit TSAPI clients & SDK	25
Chapter 2: The TSAPI Programming Environment	27
Contents of the TSAPI SDK	28
TSAPI SDK header files.....	28
TSAPI Service client libraries.....	28
TSAPI client library configuration file (TSLIB).....	30
Code Samples (Windows client only)	31
TSAPI for Windows SDK Overview.....	33

TSAPI SDK for Linux.....	35
Basic TSAPI programming tips	38
Opening and closing streams	38
Monitoring switch object state changes	38
Client/server roles and the routing service	39
The client/server session and the operation invocation model	39
Advanced TSAPI Programming Topics	40
Transferring or conferencing a call together with screen pop information	41
CSTA Services Used to Conference or Transfer Calls	42
Using the Consultation Call Service	42
Unique Advantage of the Consultation Call Service	42
Emulating Manual Operations	42
Using Original Call Information to Pop a Screen	44
Using UUI to Pass Information to Remote Applications	46
Re-registering as a Routing Server after a TCP/IP failure.....	47
Who can benefit from this route register request feature?	47
Routing transactions	48
Server-side programming considerations	53
Multiple AE Services server considerations.....	53
CTI Link Availability.....	54
Chapter 3: Control Services	55
Control Services provided by TSAPI.....	56
API Control Services	56
CSTA Control Services	57
Opening, Closing and Aborting an ACS stream.....	57
Opening an ACS stream	58
Closing an ACS stream.....	59
Aborting an ACS stream	60
Sending CSTA Requests and Responses	61
Receiving Events	62
Blocking Event Reception	62
Non-Blocking Event Reception	63
Specifying TSAPI versions when you open a stream	64
Providing a list of TSAPI versions in the API version parameter	64
How the TSAPI version is negotiated	64
Requesting private data when you open an ACS stream	66
Querying for Available Services	66
ACS functions and confirmation events	67
acsOpenStream().....	68
ACSOpenStreamConfEvent	75
acsCloseStream()	77
ACSCloseStreamConfEvent	79
ACSUniversalFailureConfEvent.....	81
acsAbortStream().....	83

acsGetEventBlock()	84
acsGetEventPoll()	87
acsGetFile() (Linux)	90
acsSetESR() (Windows)	91
acsEventNotify() (Windows)	93
acsFlushEventQueue()	96
acsEnumServerNames()	98
acsGetServerID()	100
acsQueryAuthInfo()	101
acsSetHeartbeatInterval()	104
ACSSetHeartbeatIntervalConfEvent	106
acsErrorString()	108
acsReturnCodeString()	109
acsReturnCodeVerboseString()	110
ACS Unsolicited Events	111
ACSUniversalFailureEvent	111
ACS Data Types	115
ACS Common Data Types	116
ACS Event Data Types	121
CSTA control services and confirmation events	122
cstaGetAPICaps()	123
CSTAGetAPICapsConfEvent	125
cstaGetDeviceList()	128
CSTAGetDeviceListConfEvent	130
cstaQueryCallMonitor()	132
CSTAQueryCallMonitorConfEvent	133
cstaErrorString()	135
CSTA Event Data Types	136
Chapter 4: CSTA Service Groups supported by the TSAPI Service	142
Supported Services and Service Groups	143
CSTA Objects	148
The CSTA Device object	148
Device Type	148
Device Class	149
Device Identifier	149
Device History	155
The CSTA Call object	157
The CSTA Connection object	158
CSTAUniversalFailureConfEvent	164
Chapter 5: Avaya TSAPI Service Private Data	165
What is private data?	175
What is a private data version?	176
Linking your application to the private data functions	177
Summary of TSAPI Service Private Data	178

Private Data Version 17 features	182
Service Observe Filter	182
Service Observe state change event	182
Service Observe Activation	183
Service Observe Deactivation	183
Service Observe Set connection mode	183
Query Service observe status	183
Private Data Version 15 features	186
Channel Type in Originated Event	186
VideoCall Type in MakeCall	186
VideoCall Type in AnswerCall	186
VideoCall Type in ConsultationCall	186
VideoCall Type in SingleStepTransferCall	187
SkillInfoList in QueryAgentLoginResp	187
SkillInfoList in LoggedOnEvent	188
Reserverd Data Type in RouteRequestEvent	188
Reserverd Data Type in RouteSelect	189
Private Data Version 13 features	190
Channel Type in Delivered Event	190
Channel Type in Established Event	190
Private Data Version 12 features	191
Calling Device in Diverted Event	191
Private Data Version 11 features	192
Endpoint Registration Info Query	192
Endpoint Registered Event	194
Endpoint Unregistered Event	194
Failed Event Device Identifiers	195
Private Data Version 10 Features	197
Agent Work Mode in Logged On Event	197
Consultation Call Support for Consult Options	197
Enhanced Station Status Query	198
Private Data Version 9 Features	200
Consult Mode for Held, Service Initiated, and Originated Events	200
UCID in Single Step Transfer Call Confirmation Event	201
Private Data Version 8 Features	202
Single Step Transfer Call	202
Calling Device in Failed Event	202
Requesting private data	203
Sample code for requesting private data	204
Applications that do not use private data	205
CSTA Get API Capabilities confirmation structures for Private Data Version 11 and later	206
Get API Capabilities Private Data Version 11 and Later Syntax	207
Private Data Service sample code	208
Upgrading and maintaining applications that use private data	217

Using the private data header files.....	219
The attpdefs.h file – PDU names and numbers.....	219
The attrpriv.h file – other related PDU elements.....	220
Upgrading PDV 13 applications to PDV 15	222
Upgrading PDV 12 applications to PDV 13	224
Upgrading PDV 11 applications to PDV 13	226
Channel Type in Delivered Event	Error! Bookmark not defined.
Channel Type in Established Event.....	Error! Bookmark not defined.
Upgrading PDV 10 applications to PDV 13	227
Upgrading PDV 9 applications to PDV 13	229
Upgrading PDV 8 applications to PDV 13	231
Upgrading PDV 7 applications to PDV 13	233
Upgrading PDV 6 applications to PDV 13	234
Maintaining applications that use prior versions of private data	235
Maintaining a PDV 12 application in a PDV 13 environment	235
Maintaining a PDV 11 application in a PDV 13 environment	235
Maintaining a PDV 10 application in a PDV 13 environment	237
Maintaining a PDV 9 application in a PDV 13 environment	239
Maintaining a PDV 8 application in a PDV 13 environment	240
Maintaining a PDV 7 application in a PDV 13 environment	241
Recompiling against the same SDK	242
Chapter 6: Call Control Service Group	243
Graphical Notation Used in the Diagrams.....	245
Alternate Call Service	246
Answer Call Service	246
Clear Call Service	247
Clear Connection Service	247
Conference Call Service	248
Consultation Call Service.....	248
Consultation Direct-Agent Call Service.....	249
Consultation Supervisor-Assist Call Service.....	249
Deflect Call Service.....	250
Hold Call Service.....	250
Make Call Service	250
Make Direct-Agent Call Service	251
Make Predictive Call Service	251
Make Supervisor-Assist Call Service	252
Pickup Call Service	252
Reconnect Call Service.....	252
Retrieve Call Service	253
Single Step Conference Call.....	253
Single Step Transfer Call.....	254
Transfer Call Service	254
Alternate Call Service.....	255

Answer Call Service	259
Clear Call Service	265
Clear Connection Service	267
Conference Call Service	276
Consultation Call Service	283
Consultation Direct-Agent Call Service	297
Consultation Supervisor-Assist Call Service	307
Deflect Call Service	317
Hold Call Service	322
Make Call Service	326
Make Direct-Agent Call Service	342
Make Predictive Call Service	352
Make Supervisor-Assist Call Service	364
Pickup Call Service	373
Reconnect Call Service	378
Retrieve Call Service	387
Send DTMF Tone Service (Private Data Version 4 and Later)	391
Selective Listening Hold Service (Private Data Version 5 and Later)	399
Selective Listening Retrieve Service (Private Data Version 5 and Later)	405
Single Step Conference Call Service (Private Data Version 5 and Later)	410
Single Step Transfer Call (Private Data Version 8 and later)	419
Transfer Call Service	427
Chapter 7: Set Feature Service Group	434
Set Advice of Charge Service (Private Data Version 5 and Later)	435
Set Agent State Service	439
Set Billing Rate Service (Private Data Version 5 and Later)	452
Set Do Not Disturb Feature Service	457
Set Forwarding Feature Service	460
Set Message Waiting Indicator (MWI) Feature Service	464
Chapter 8: Query Service Group	468
Query ACD Split Service	469
Query Agent Login Service	473
Query Agent State Service	483
Query Call Classifier Service	493
Query Device Info	497
Query Device Name Service	504
Query Do Not Disturb Service	511
Query Endpoint Registration Info Service (Private Data Version 11 and later)	514
Query Forwarding Service	522
Query Message Waiting Indicator Service	526
Query Station Status Service	531
Query Time of Day Service	536
Query Trunk Group Service	540
Query Universal Call ID Service (Private)	544

Chapter 9: Snapshot Service Group	548
Snapshot Call Service	549
Snapshot Device Service	556
Chapter 10: Monitor Service Group.....	564
Overview	564
Change Monitor Filter Service — cstaChangeMonitorFilter()	564
Monitor Call Service — cstaMonitorCall()	564
Monitor Calls Via Device Service — cstaMonitorCallsViaDevice().....	564
Monitor Device Service — cstaMonitorDevice()	564
Monitor Ended Event — CSTAMonitorEndedEvent	565
Monitor Stop On Call Service (Private) — attMonitorStopOnCall()	565
Monitor Stop Service — cstaMonitorStop()	565
Event Filters and Monitor Services	565
The localConnectionInfo Parameter for Monitor Services	569
Change Monitor Filter Service.....	570
Monitor Call Service	578
Monitor Calls Via Device Service	591
Special Rules – Monitor Calls Via Device Service.....	595
Monitor Device Service	603
Monitor Ended Event Report.....	614
Monitor Stop On Call Service (Private)	616
Monitor Stop Service.....	620
Chapter 11: Event Report Service Group	623
CSTAEventCause and LocalConnectionState.....	625
Event Minimization Feature on Communication Manager	632
Call Cleared Event	633
Charge Advice Event (Private).....	639
Conferenced Event	644
Connection Cleared Event	668
Delivered Event.....	678
Diverted Event.....	725
Do Not Disturb Event	735
Endpoint Registered Event (Private Data Version 11 and later).....	737
Endpoint Unregistered Event (Private Data Version 11 and later)	744
Entered Digits Event (Private).....	751
Established Event	754
Failed Event	789
Forwarding Event	799
Held Event.....	802
Logged Off Event	807
Logged On Event	810
Message Waiting Event	815
Network Reached Event	818
Originated Event	827

Queued Event	838
Retrieved Event.....	845
Service Initiated Event	848
Transferred Event	854
Event Report Detailed Information	877
Analog Sets.....	877
Redirection.....	877
Redirection on No Answer	877
Switch Hook Operation	877
ANI Screen Pop Application Requirements	878
Announcements	878
Answer Supervision	878
Attendants and Attendant Groups.....	879
Attendant Specific Button Operation.....	879
Attendant Auto-Manual Splitting	880
Attendant Call Waiting	880
Attendant Control of Trunk Group Access	880
AUDIX	880
Automatic Call Distribution (ACD).....	881
Announcements.....	881
Interflow	881
Night Service	881
Service Observing	881
Auto-Available Split	881
Bridged Call Appearance	881
Busy Verification of Terminals	882
Call Coverage	882
Call Coverage Path Containing VDNs	883
Call Forwarding All Calls.....	883
Call Park.....	883
Call Pickup	884
Call Vectoring.....	884
Call Prompting	886
Lookahead Interflow	886
Multiple Split Queuing.....	886
Call Waiting.....	886
Conference.....	886
Consult Button.....	886
CTI Link Failure.....	887
Data Calls.....	887
DCS.....	887
Direct Agent Calling and Number of Calls In Queue	887
Drop Button Operation	887
Expert Agent Selection (EAS).....	888

Logical Agents	888
Hold	888
Integrated Services Digital Network (ISDN)	888
Multiple Split Queuing	889
Personal Central Office Line (PCOL)	889
Primary Rate Interface (PRI)	889
Ringback Queuing	890
Send All Calls (SAC)	890
Service-Observing	890
Temporary Bridged Appearances	890
Terminating Extension Group (TEG)	890
Transfer	891
Trunk-to-Trunk Transfer	891
Chapter 12: Routing Service Group	892
Route End Event	893
Route End Service (TSAPI Version 2)	898
Route End Service (TSAPI Version 1)	901
Route Register Abort Event	903
Route Register Cancel Service	905
Route Register Service	908
Route Request Event (TSAPI Version 2)	911
Route Request Event (TSAPI Version 1)	932
Route Select Service (TSAPI Version 2)	936
Route Select Service (TSAPI Version 1)	952
Route Used Event (TSAPI Version 2)	954
Route Used Event (TSAPI Version 1)	958
Chapter 13: System Status Service Group	961
Overview	961
System Status Request Service – cstaSysStatReq()	961
System Status Start Service – cstaSysStatStart()	961
System Status Stop Service – cstaSysStatStop()	961
Change System Status Filter Service cstaChangeSysStatFilter()	961
System Status Event – CSTASysStatEvent	961
System Status Events – Not Supported	961
System Status Request Service	962
System Status Start Service	969
System Status Stop Service	978
Change System Status Filter Service	980
System Status Event	989
Appendix A: Universal Failure Events	996
Common switch-related CSTA Service errors	996
TSAPI Client library error codes	1003
ACSUniversalFailureConfEvent error values	1007
ACS Related Errors	1025

Appendix B: Summary of Private data support	1027
Private Data Version 17 features	1027
Private Data Version 16 features	1027
Private Data Version 15 features	1027
Channel Type in Originated event	1027
VideoCall Option	1027
Private Data Version 14 features	1027
Private Data Version 13 features	1027
Channel Type in Delivered Event	1028
Channel Type in Established Event	1028
Private Data Version 12 features	1028
Calling Device in Diverted Event	1028
Private Data Version 11 features	1028
Endpoint Information Query	1028
Endpoint Registered Event	1028
Endpoint Unregistered Event	1029
Failed Event Device Identifiers	1029
Private Data Version 11 features, services, and events	1029
Private Data Version 10 features	1030
Agent Work Mode in Logged On Event	1030
Consult Options	1030
Enhanced Station Status Query	1031
Private Data Version 10 features, services, and events	1031
CSTA Get API Capabilities Private Data Versions 8 – 10 Syntax	1032
Private Data Version 9 features	1033
Consult Modes	1033
UCID in Single Step Transfer Call Confirmation event	1033
Private Data Version 9 features, services, and events	1034
Private Data Version 8 features	1035
Single Step Transfer Call Escape Service	1035
Calling Device in Failed Event	1035
Private Data Version 8 features, services, and events	1035
Private Data Version 7 features	1036
Network Call Redirection for Routing	1036
Redirecting Number Information Element (presented through DeviceHistory)	1036
Query Device Name for Attendants	1036
CSTA Get API Capabilities for Private Data Version 7	1037
Increased Aux Reason Codes	1037
Private Data Version 7 features, services, and events	1038
CSTA Get API Capabilities confirmation structures for Private Data Version 7	1039
Private Data Version Feature Support prior to AE Services TSAPI R3.1.0	1041
Summary of private data versions 2 through 6	1042
CSTA Device ID Type (Private Data Version 4 and Earlier)	1049
CSTAGetAPICaps Confirmation interface structures for Private Data Versions 4, 5, and 6	1050

Get API Capabilities Private Data Version 5 and 6 Syntax.....	1051
Get API Capabilities Private Data Version 4 Syntax.....	1051
Private Data Function Changes between V5 and V6.....	1052
Private Data Sample Code.....	1053
Appendix C: Server-Side Capacities.....	1061
Communication Manager CSTA system capacities.....	1062
Index	1067

About this document

This document, the *Avaya Aura® Application Enablement Services TSAPI for Avaya 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](#) provides a brief overview of the AE Services TSAPI Service.
- [Chapter 2: The TSAPI Programming Environment](#) 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](#) 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](#) describes the CSTA Service groups that the TSAPI Service supports.
- [Chapter 5: Avaya TSAPI Service Private Data](#) 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](#) 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](#) 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](#) 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](#) 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](#) describes the three types of monitor services the TSAPI Service provides for Communication Manager.
- [Chapter 11: Event Report Service Group](#) describes event messages (or reports) from Communication Manager to the TSAPI Service.
- [Chapter 12: Routing Service Group](#) describes the services that allow the switch to request and receive routing instructions for a call.

- [Chapter 13: System Status Service Group](#) describes the services that allow an application to receive reports on the status of the switching system.
- [Appendix A: Universal Failure Events](#) describes ACS Universal Failure Events.
- [Appendix B: Summary of Private data support](#) describes previous private data versions of AE Services.
- [Appendix C: Server-Side Capacities](#) describes server-side capacities, which include Avaya Communication Manager capacities and AE Services TSAPI Service capacities.

What's New for the AE Services Release 8.1.11 TSAPI Service

For AE Services Release 8.1.11, the TSAPI Service supports the following enhancements to AE Services server:

- Added validation for Key Usage and Extended Key Usage in AES server certificate during TLS handshake between AES and TSAPI Client over secure link connection.

What Was New in Earlier Releases of the TSAPI Service

This section highlights changes to the TSAPI for Communication Manager Programmer's Reference for AE Services Releases 5.0 through 8.1.3:

What's New for the AE Services Release 8.1.3 TSAPI Service

For AE Services Release 8.1.3, the TSAPI Service supports the following enhancements to AE Services server:

- 64-bit TSAPI clients & SDK support
- Added validation for WebLM server hostname in WebLM server identity certificate's SAN or CN field during TLS handshake
- Service Observe(SO) via CTI support - for activation /deactivation /mode change /query SO via CTI. Also, monitor APIs are enhanced to support SO state change events on endpoint.

NOTE: ASAI Link Version 12 or above required to subscribe for these events.

What's New for the AE Services Release 8.1.2 TSAPI Service

For AE Services Release 8.1.2, the TSAPI Service supports the following enhancements to AE Services server:

- Increased conference party limit from 6 to 12.
- Increased bridging support from 20 to 255 for ASAI interaction.

NOTE: ASAI Link Version 11 and above required to avail above features.

What's New for the AE Services Release 8.1 TSAPI Service

For AE Services Release 8.1, the TSAPI Service supports the following enhancements to AE Services server:

Private Data Version 15

AE Services 8.1 introduces private data version 15, which add support for Channel Type in CSTA OriginatedEvent, add VideoCall Type feature in

MakeCall, AnswerCall, ConsultationCall and SingleStepTransfer.VideoCall option only support for Avaya Aura Media Server(AMS)

Originated event Channel Type values

CH_OTHER - The channel type is other than voice or video.
CH_UNKNOWN - The channel type is not specified.
CH_VOICE - The channel type is voice.
CH_VIDEO - The channel type is video.

VideoCall Type values

VID_DISABLED – Video Call is disable
VID_INCOMING – Video Call is one way only incoming video
VID_TWOWAY – Video Call enable from both side

What's New for the AE Services Release 7.1.1 TSAPI Service

For AE Services Release 7.1.1, the TSAPI Service supports the following enhancements to AE Services server:

- AE Services 7.1.1 has added support for Channel Type in CSTA Delivered and CSTA Established events.

Delivered and Established event Channel Type values

CH_OTHER - The channel type is other than voice or video.
CH_UNKNOWN - The channel type is not specified.
CH_VOICE - The channel type is voice.
CH_VIDEO - The channel type is video.

What's New for the AE Services Release 7.0 TSAPI Service

For AE Services Release 7.0, the TSAPI Service supports the following enhancements to AE Services server:

- AE Services 7.0.x runs on Red Hat Enterprise Linux version 6 update 5 (RHEL 6U5), with appropriate security patches from Red Hat.
- Support for the TLSv1.2 protocol on AE Services server 7.0.1
Use of a new 1-year self-signed server certificate for the AE Services server as the default server certificate

Changes to the Application Enablement Services offer

- Removed support for the SSLv3 protocol

What's New for the AE Services Release 6.3.3 TSAPI Service

CSTA Message Waiting Events

Beginning with AE Services Release 6.3.3, Avaya Communication Manager Release 6.3.2, and ASAI Link Version 5, the TSAPI Service supports the CSTA Message Waiting event. This event indicates that a monitored station's message waiting indicator has been activated or deactivated. See the [Private Data Version 14](#)

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>

/* ATTV14LoggedOnEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_LOGGED_ON */
    union
    {
        ATTV14LoggedOnEvent_t    loggedOnEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV14LoggedOnEvent_t {
    ATTWorkMode_t    workMode;
} ATTV14LoggedOnEvent_t;

typedef enum ATTWorkMode_t {
    WM_NONE = -1,
    WM_AUX_WORK = 1,
    WM_AFTCAL_WK = 2,
    WM_AUTO_IN = 3,
    WM_MANUAL_IN = 4
} ATTWorkMode_t;
```

Message Waiting Event on page 814.

Undelivered Call Events

Beginning with AE Services Release 6.3.3, Avaya Communication Manager Release 6.3.6, and ASAI Link Version 7, the TSAPI Service provides CSTA Unsolicited events to indicate that a call could not be delivered to a monitored station. For example:

- If a call receives Call Coverage treatment without alerting at the called station, a device monitor for the called station receives a CSTA [Diverted Event](#).
- If a call is forwarded without alerting at the called station, a device monitor for the called station receives a CSTA [Diverted Event](#).
- If a call cannot be completed because the called station is not registered, a device monitor for the called station receives a CSTA [Failed Event](#).
- If a call cannot be completed because the called station does not have a call appearance available to receive the call or because the Limit Number of Concurrent Calls feature is active, a device monitor for the called station receives a CSTA [Failed Event](#).

CSTA Diverted Event Cause Values

Prior to AE Services Release 6.3.3, the only supported cause value for the CSTA [Diverted Event](#) was EC_REDIRECTED. Beginning with AE Services Release 6.3.3 and Avaya Communication Manager Release 6.3.6, the following event cause values are also supported:

- EC_CALL_FORWARD_ALWAYS – The call was redirected because Call Forwarding is active at the diverting device.
- EC_CALL_FORWARD – The call received Call Coverage treatment because Cover All Calls is active at the diverting device.
- EC_CALL_FORWARD_BUSY – The call received Call Coverage treatment because the diverting device was busy or did not have a call appearance available to receive the call.
- EC_CALL_FORWARD_NO_ANSWER – The call received Call Coverage treatment because the diverting device did not answer the call within the administered number of rings.

Private Data Version 12

AE Services Release 6.3.3 introduces private data version 12, which adds the calling device to the private data accompanying the CSTA Diverted Event. See [Calling Device in Diverted Event](#) on page 191.

21-Digit Telephone Numbers

Beginning with AE Services Release 6.3.3, Avaya Communication Manager Release 6.3.6, and ASAI Link Version 7, the calling party number, connected number, and redirecting number information elements in ASAI events may contain up to 21 digits. For ASAI Link Versions 1-6, these information elements contained a maximum of 15 digits, so longer telephone numbers were truncated. The increased number of digits is reflected in TSAPI events.

What's New for the AE Services Release 6.3.1 TSAPI Service

Private Data Version 11

AE Services Release 6.3.1 introduces private data version 11, which includes the following features:

- Endpoint Registration Information Query – This new query provides information about the H.323 and SIP endpoints registered to a station extension number. See [Query Endpoint Registration Info Service \(Private Data Version 11 and later\)](#) on page 514.
- Endpoint Registered Event – This new CSTA Private Status Event indicates that an H.323 or SIP endpoint has registered to a monitored station extension. See [Endpoint Registered Event \(Private Data Version 11 and later\)](#) on page 737.
- Endpoint Unregistered Event – This new CSTA Private Status Event indicates that an H.323 or SIP endpoint has unregistered from a monitored station extension. See [Endpoint Unregistered Event \(Private Data Version 11 and later\)](#) on page 744.
- CSTA Failed Event Device IDs – Beginning with private data version 11, the DeviceID components of the `failingDevice` and `failedConnection` parameters in the CSTA [Failed Event](#) are set differently than for earlier private data versions.
- New Get API Capabilities confirmation event – see [CSTA Get API Capabilities confirmation structures for Private Data Version 11 and later](#) on page 206.

What's New for the AE Services Release 6.2.0 TSAPI Service

Private Data Version 10

AE Services Release 6.2.0 introduces private data version 10, which includes the following features:

- Agent Work Mode in Logged On Event – Beginning with private data version 10, private data accompanying a CSTA [Logged On Event](#) provides the initial work mode of the agent.
- Consult Options – Applications may use the new private data formatting function `attV10ConsultationCall()` to indicate the intended purpose of a consultation call. The specified Consult Options are reflected in the private data associated with the CSTA [Held Event](#), [Service Initiated Event](#), and [Originated Event](#) resulting from the consultation call.
- Enhanced Station Status Query – A new version of the station status query, `attV10QueryStationStatus()`, queries for a station's service state in addition to its talk state.

What's New for the AE Services Release 6.1.0 TSAPI Service

Private Data Version 9

AE Services Release 6.1.0 introduces private data version 9, which includes the following features:

- Consult Modes – In some scenarios, private data associated with the CSTA [Held Event](#), [Service Initiated Event](#), and [Originated Event](#) will now indicate whether these events are related to a conference operation, a transfer operation, or a Consultation Call service request.
- The Single Step Transfer Call Confirmation Event now includes the Universal Call ID (UCID) of the resulting call.

What's New for the AE Services Release 5.2.0 TSAPI Service

String Functions

AE Services Release 5.2.0 adds several functions to convert TSAPI function return codes, ACS Universal Failure error values, and CSTA Universal Failure error values to strings:

- [acsErrorString\(\)](#) returns a pointer to a string describing an ACSUniversal-Failure_t error value.
- [acsReturnCodeString\(\)](#) returns a pointer to a terse string describing a RetCode_t return code value.
- [acsReturnCodeVerboseString\(\)](#) returns a pointer to a terse string describing a RetCode_t return code value.
- [cstaErrorString\(\)](#) returns a pointer to a string describing a CSTAUniversal-Failure_t error value.

What's New for the AE Services Release 4.1.0 TSAPI Service

CSTA Do Not Disturb Events

Beginning with AE Services Release 4.1.0, Avaya Communication Manager Release 5.0, and ASAI Link Version 5, the TSAPI Service supports the CSTA [Do Not Disturb Event](#). This event indicates that the status of the Send All Calls feature has changed for a monitored station extension.

CSTA Forwarding Events

Beginning with AE Services Release 4.1.0, Avaya Communication Manager Release 5.0, and ASAI Link Version 5, the TSAPI Service supports the CSTA [Forwarding Event](#). This event indicates that the status of the Forwarding feature has changed for a monitored station extension.

Private Data Version 8

AE Services 4.1.0 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 419.
- Calling Device in Failed Event – see [Calling Device in Failed Event](#) on page 202.
- New Get API Capabilities confirmation event – [see CSTA Get API Capabilities Private Data Versions 8 – 10 Syntax](#) on page 206.
- A new private data parameter, flowPredictiveCallEvents, has been added to the CSTAMonitorCallsViaDevice service. For more information, see [Monitor Calls Via Device Service](#) on page 591.

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.

For 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 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 68.

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) Management Console, as follows:

- (Optional) Administer a range of "Encrypted TLINK Ports" on the Ports OAM page. To access this setting from the Application Enablement Services Management Console, select Networking > Ports.
- Administer TSAPI links with the Encrypted security setting on the Add / Edit TSAPI Links OAM page. To access this page from the Application Enablement Services Management Console, select AE Services > TSAPI > TSAPI Links. From the TSAPI Links OAM page, select Add Link or Edit Link.

For more information, see the *Avaya Aura® Application Enablement Services Administration and Maintenance Guide*, 02-300357 and the AE Services Management Console Help pages.

TSAPI client heartbeat

Beginning with AE Services Release 4.1.0, the TSAPI Service automatically provides a client heartbeat, allowing the client library to detect network connectivity issues between the AE Services server and the client. For more information, see the following topics:

- [Opening an ACS stream](#) on page 58
- [acsOpenStream\(\)](#) on page 68.
- [acsSetHeartbeatInterval\(\)](#) on page 104.

Alternate TLINK capability

Beginning with AE Services Release 4.1.0, the TSAPI Service provides applications with the ability to specify an optional list of alternate TLINKs to be used automatically and transparently by `acsOpenStream()`. The alternate TLINKs are only used if the TLINK specified in the open stream call is not available at the time that procedure was invoked.

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 100.

What's New for the AE Services 4. 0 TSAPI Service

Route Registration Request service update

Beginning with AE Services 4.0, the TSAPI Service allows 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 48.
- [Route Register Service](#) on page 908.
- [Route Register Abort Event](#) on page 903.

AE Services 8.1.3 clients and backward compatibility

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

AE Services Release 8.1.3 supports the AE Services Release 8.1.2 TSAPI clients and is backward compatible with AE Services TSAPI clients for Releases 8.1.2, 8.1.1, 8.1, 8.0, 7.1.3, 7.1.2, 7.0.0, 6.3.3, 6.3.1, 6.3.0, 6.2.x and 6.1.x.

In AE Services 8.1.3, only the Transport Layer Security (TLS) 1.2 protocol is enabled by default. The lower level TLS protocols 1.0 and 1.1 are disabled by default. Note, according to the National Institute of Standards and Technology (NIST) Special Publication 800-52, TLS version 1.1 is required, at a minimum, in order to mitigate various attacks on the TLS 1.0 protocol. The use of TLS 1.2 is strongly recommended.

This change may cause older AE Services clients (version AE Services 7.0 and earlier) that are using TLS to fail to establish a secure socket connection to the AE Services 8.1.3 server. In order to achieve a more secure client/server socket connection, we encourage current client applications to use an AE Services 8.1.2 SDK where the TLS 1.2 protocol is supported.

The default certificate used by 6.x and older releases of AE Services has been replaced. Any applications that rely on this certificate for SSL/TLS will fail SSL negotiation with an AE Services 8.1.3 server. To overcome this in your test environment, you will need to export the server certificate from the AE Services and use it in your application's configuration. Note that, for a production environment, you are strongly advised to create your own server certificates and install them on the AE Services server (and include the CA in your application's trust-store). Please see the AE Services Security White Paper for more information.

About installing the SDK

This programming guide assumes that you have installed the AE Services TSAPI client and the Software Development Kit (SDK). The *Avaya Aura® Application Enablement Services TSAPI and CVLAN Client and SDK Installation Guide*, 02-300543, provides instructions for installing the TSAPI client and the SDK in Chapter 2, "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 14
- [Related Avaya documents](#) on page 15

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 Aura® Application Enablement Services TSAPI and CVLAN Client and SDK Installation Guide*, 02-300543.

Use this document to install the software development kits (SDKs) described in this document, the TSAPI for Avaya Communication Manager Programmer's Reference.

- *Avaya Aura® 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 (www.avaya.com/devconnect).
- 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 the DevConnect program.

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	<code>#include <acs.h></code>	Coding examples. Note: Coding examples contain operators and special characters that are part of the C programming language syntax. For example, the angle brackets in the example are part of the C language 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 appear in upper case, they refer to structures or manifest constants.
lowercase • acs • csta	acsOpenStream cstaQueryCallMonitor	When the terms acs and csta appear in lower case, they refer to function names.

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 <i>Avaya MultiVantage Application Enablement Services TSAPI Programmer Reference</i> , 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 <code>acsHandle</code> , <code>invokeID</code> , and <code>privateData</code> 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 <code>acsHandle</code> , <code>eventClass</code> , <code>eventType</code> , and <code>privateData</code> 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 Parameters: universalFailure	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code> . The error parameter in this event may contain one of the error values described in the CSTAUniversalFailureConfEvent on page 164.

Table 1 : Service Description page elements

Element	Description
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.
support level	Identifies the level of support for each service parameter: [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 <code>OBJECT_NOT_KNOWN</code>. [mandatory – partially supported] This parameter is mandatory as defined in Standard ECMA-179. However, the TSAPI Service 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. [mandatory – not supported] This parameter is mandatory as defined in Standard ECMA-179. However, The TSAPI Service does not support this parameter due to Communication Manager feature limitations. “Not supported” means that whether the application provides a value 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] 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. [optional – supported] This parameter is optional as defined in Standard ECMA-179, but it is always supported. [optional – partially supported] This parameter is optional as defined in Standard ECMA-179. However, the TSAPI Service 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.

Table 1 : Service Description page elements

Element	Description
	[optional – not supported] This parameter is optional as defined in Standard ECMA-179, but it is not supported by The TSAPI Service. “Not supported” means that whether the application provides a value or not, the value specified will be ignored and the TSAPI Service will assign a Communication Manager default value. [optional – limited support] This parameter is optional as defined in Standard ECMA-179, but it is not fully supported by the TSAPI Service. 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.
support level	Identifies the level of support for each private parameter: [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] This parameter is optional for the specific service. It may or may not be present in the private data. If not, the TSAPI Service will assign a Communication Manager default value. [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.
Ack Private Parameters:	

Table 1 : Service Description page elements

Element	Description
parameter	Detailed information for each parameter in the private data of the service confirmation event.
Nak Parameters:	
universalFailure	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code> . The <code>error</code> parameter in this event may contain one of the error values described in the CSTAUniversalFailureConfEvent on page 164.
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 .
Private Data 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 is 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 contains the following topics:

- [Introduction](#) on page 23
- [Ecma International and the CSTA Standards](#) on page 24
- [The TSAPI Specification](#) on page 24
- [TSAPI for Avaya Communication Manager](#) on page 24
- [The TSAPI Client](#) on page 25
- [The TSAPI SDK](#) on page 25
- [64 bit TSAPI clients & SDK](#)

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 Manager. 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 Aura® Application Enablement Services TSAPI for Avaya 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 Avaya Communication Manager Programmer Reference*, 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 does not include the TSAPI Client. For more information about the TSAPI SDK, see [Contents of the TSAPI SDK](#) on page 28.

64 bit TSAPI clients & SDK

Starting from 8.1.3 , AES has added support for 64-bit TSAPI clients and SDK. Following header changes have been made to add the support.

```
Item # 1 :-
Replace
"typedef unsigned long InvokeID_t;"
With
"typedef unsigned int InvokeID_t;"

Item # 2:-
Replace
typedef unsigned short ACSHandle_t;
With

Windows :-
typedef __int64 ACSHandle_t;
Linux :-
typedef unsigned long ACSHandle_t;

Item # 3 :-
Replace :-
typedef Boolean (*EnumServerNamesCB) (
char *serverName,
unsigned long lParam);
RetCode_t acsEnumServerNames(
StreamType_t streamType, /* INPUT */
EnumServerNamesCB callback, /* INPUT */
unsigned long lParam); /* INPUT */
with
Linux :-
typedef Boolean (*EnumServerNamesCB) (
char *serverName,
```

```

unsigned long lParam);
RetCode_t acsEnumServerNames(
StreamType_t streamType, /* INPUT */
EnumServerNamesCB callback, /* INPUT */
unsigned long lParam); /* INPUT */
Windows :-
typedef Boolean (*EnumServerNamesCB) (
char *serverName,
LPARAM lParam);
RetCode_t acsEnumServerNames(
StreamType_t streamType, /* INPUT */
EnumServerNamesCB callback, /* INPUT */
LPARAM lParam); /* INPUT */

```

Item # 4:-

```

typedef void pascal (*EsrFunc)(unsigned long esrParam)
RetCode_t acsSetESR(
ACSHandle_t acsHandle, /* INPUT */
EsrFunc esr, /* INPUT */
unsigned long esrParam, /* INPUT */
Boolean notifyAll); /* INPUT */
With
Windows :-
typedef __int64 ACSHandle_t;
typedef __int64 tsapi_uintptr_t;
typedef void pascal (*EsrFunc)( tsapi_uintptr_t esrParam)
RetCode_t acsSetESR(
ACSHandle_t acsHandle, /* INPUT */
EsrFunc esr, /* INPUT */
tsapi_uintptr_t esrParam, /* INPUT */
Boolean notifyAll); /* INPUT */

```


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:

- [Contents of the TSAPI SDK](#) on page 28
- [TSAPI SDK header files](#) on page 28
- [TSAPI Service client libraries](#) on page 28
- [TSAPI client library configuration file \(TSLIB\)](#) on page 30
- [TSAPI for Windows SDK Overview](#) on page 33
- [TSAPI SDK for Linux](#) on page 35
- [Basic TSAPI programming tips](#) on page 38
- Advanced TSAPI Programming Topics [Advanced TSAPI Programming Topics](#) on page 40
- [Server-side programming considerations](#) on page 53

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 16.

Contents of the TSAPI SDK

The AE Services TSAPI SDK consists of the following components.

- headers and libraries
- sample code
- the TSAPI Exerciser (for Windows-based clients only)

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 and CVLAN Client and SDK Installation Guide*, 02-300543.

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 219.

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.

Table 2: TSAPI Service client libraries

Library name	Operating system	Description
csta32.dll (for 32 bits arch.) or csta64.dll (for 64 bits arch.)	Windows	Contain TSAPI functions (<i>cstaServiceName</i> and the <i>acsServiceName</i>) services.
libcsta.so	Linux	
ATTPRIV32.DLL (for 32 bits arch.) or ATTPRIV64.DLL (for 64 bits arch.)	Windows	Contain private data decoding (<i>attPrivateData</i>) and encoding functions (<i>attServiceName</i>).
libattpriv.so	Linux	

<code>aes-libeay32.dll</code>	Windows	Along with <code>aes-ssleay32.dll</code> , allows the TSAPI Windows client to create secure connections to the AE Services server.
<code>aes-ssleay32.dll</code>	Windows	Along with <code>aes-libeay32.dll</code> , allows the TSAPI Windows client to create secure connections to the AE Services server.

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 Aura® Application Enablement Services TSAPI 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 sample 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 call 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 and transfer call handling	One device, many calls - Adds conferencing and transferring to Sample 3. - Includes tracking of multiple connections on a single call.

Table 4: TSAPI Sample code -- common files

File Name	Description
• TSAPI.CPP • TSAPI.H	• Helper classes for tracking devices and calls. Includes routines for retrieving events from the CSTA32.DLL (for 32 bits arch.) or CSTA64.DLL (64 bits arch.).
• 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 header files • These MFC files 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 TSAPIIN 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 TSAPIMUL application • Initialization and resources for Sample 3.

• TSAPICNF.C PP • TSAPICNF.H • TSAPICNF.R C	• Defines the class behaviors for Sample 4, the TSAPCNF application • Main header file for the TSAPICNF 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](#).

Development Platforms

AE Services requires that you use Microsoft Visual C++ 10.0 or Microsoft Visual C++ 2010 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` (for 32 bits arch.) or `CSTA64.DLL` (64 bits arch.). Specify the `CSTA32.LIB` (for 32 bits arch.) or `CSTA64.LIB` (64 bits arch.) import library when compiling your application.



NOTE:

Applications using private data should also specify the `ATTPRIV32.LIB` (for 32 bits arch.) or `ATTPRIV64.LIB` (64 bits arch.) import library.

Using Application Control Services

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

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](#).

Development Platforms

The TSAPI header files in this SDK are compatible with the GNU Linux C 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.

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 `CSTATTRACE` 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.

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_t acsHandle;
    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... */
    }
}
```

Basic TSAPI programming tips

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

- [Opening and closing streams](#) on page 38
- [Monitoring switch object state changes](#) on page 38
- [Client/server roles and the routing service](#) on page 39
- [The client/server session and the operation invocation model](#) on page 39

For more information about designing applications see [Advanced TSAPI Programming Topics](#) on page 40.

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 55.

- 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 206).
- An application should open only one stream per advertised service. Each stream carries messages for the application to one advertised service.

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, the 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.

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.

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. At that time, the application may use the `invokeID` in some other request.

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-reference ID in their acknowledgment. The cross-reference ID 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 Topics

This section discusses programming topics that are useful for designing TSAPI applications. Topics include:

- [Transferring or conferencing a call together with screen pop information](#) on page 41
- [CSTA Services Used to Conference or Transfer Calls](#) on page 42
- [Using Original Call Information to Pop a Screen](#) on page 44
- [Using UUI to Pass Information to Remote Applications](#) on page 46
- [Re-registering as a Routing Server after a TCP/IP failure](#) on page 47

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” (Dialed Number Identification Service) information 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 UUI 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](#), 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 42
- [Unique Advantage of the Consultation Call Service](#) on page 42

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, `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](#)). Next, the Conference Call or Transfer Call service 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 the transferring party hangs up before the consultation call is answered. This is known as an “unsupervised transfer”.

Notice that when the consultation party answers the unsupervised transfer, there are two parties on the call, the original caller and the consultation party. Therefore, when the calling party answers, the 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 an unsupervised 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](#) 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” cannot 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.

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

- 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:

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 on multiple Communication Manager servers
- reside on a CTI platform at a remote switch that is monitoring stations 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 benefit from this route register request feature?

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 48.

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 for the call. A transaction is completed when the TSAPI Service 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 49.

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 for a routing device

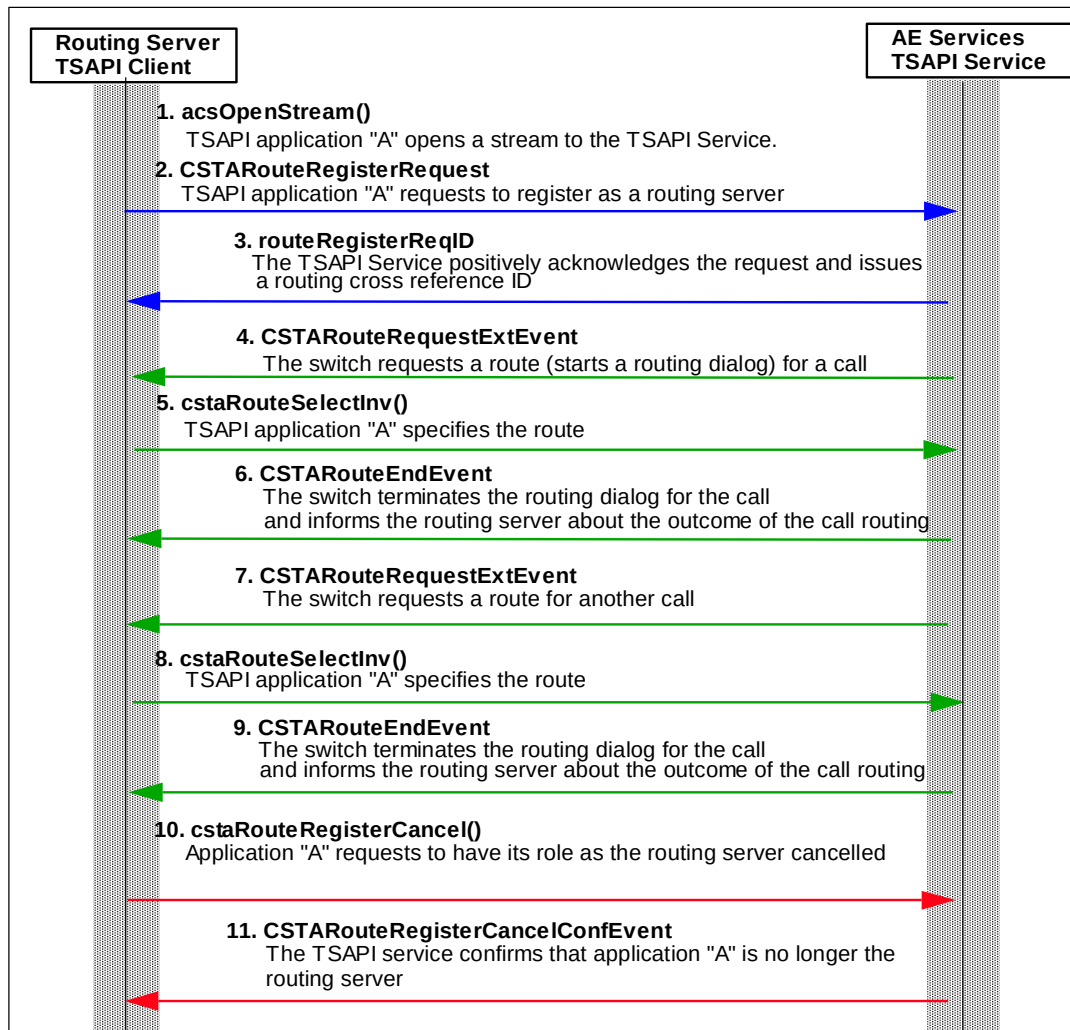
NOTE:

The TSAPI Service allows only one application to register as the routing server for a specific 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 51 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 52.

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-end TCP/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"

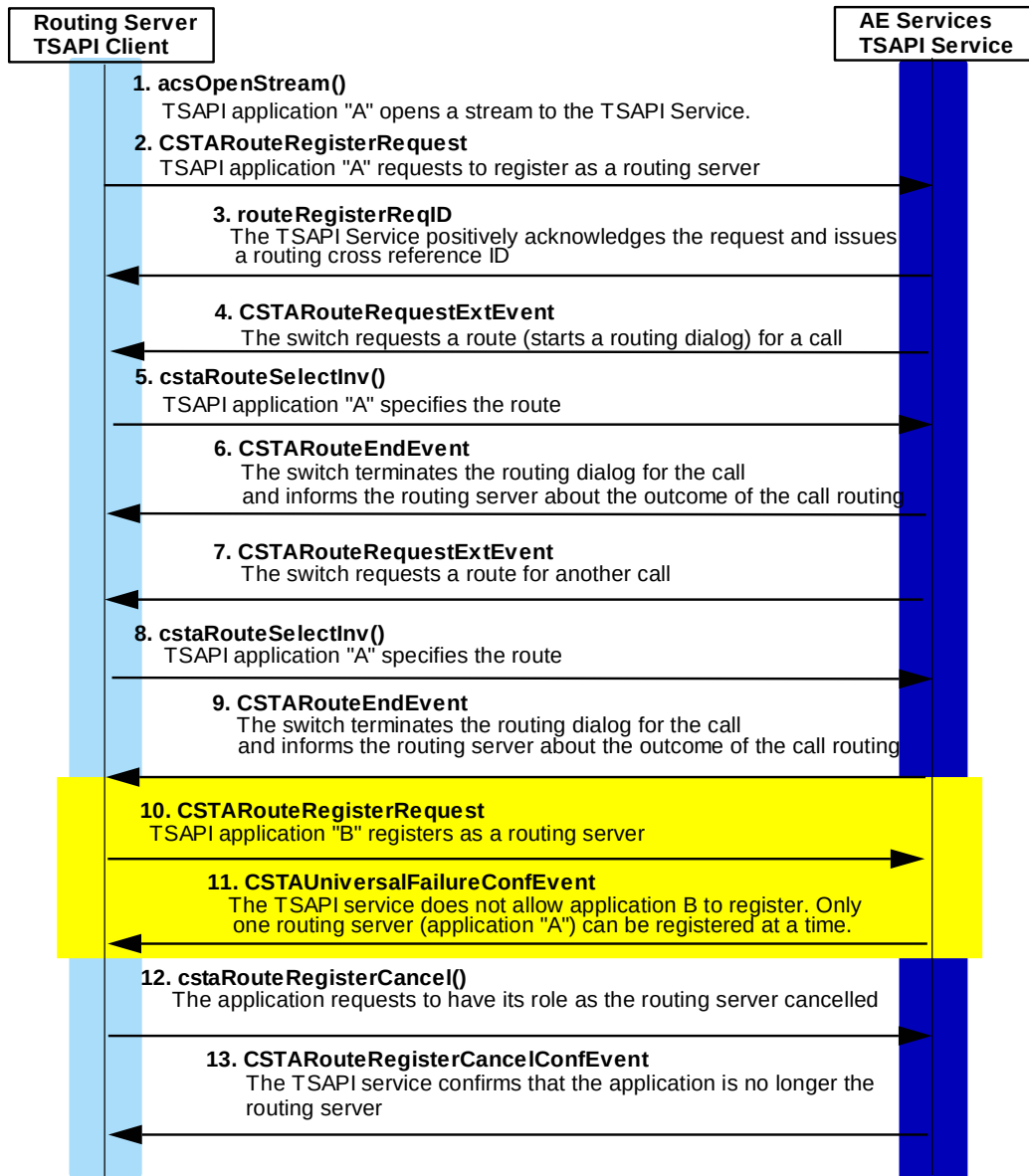
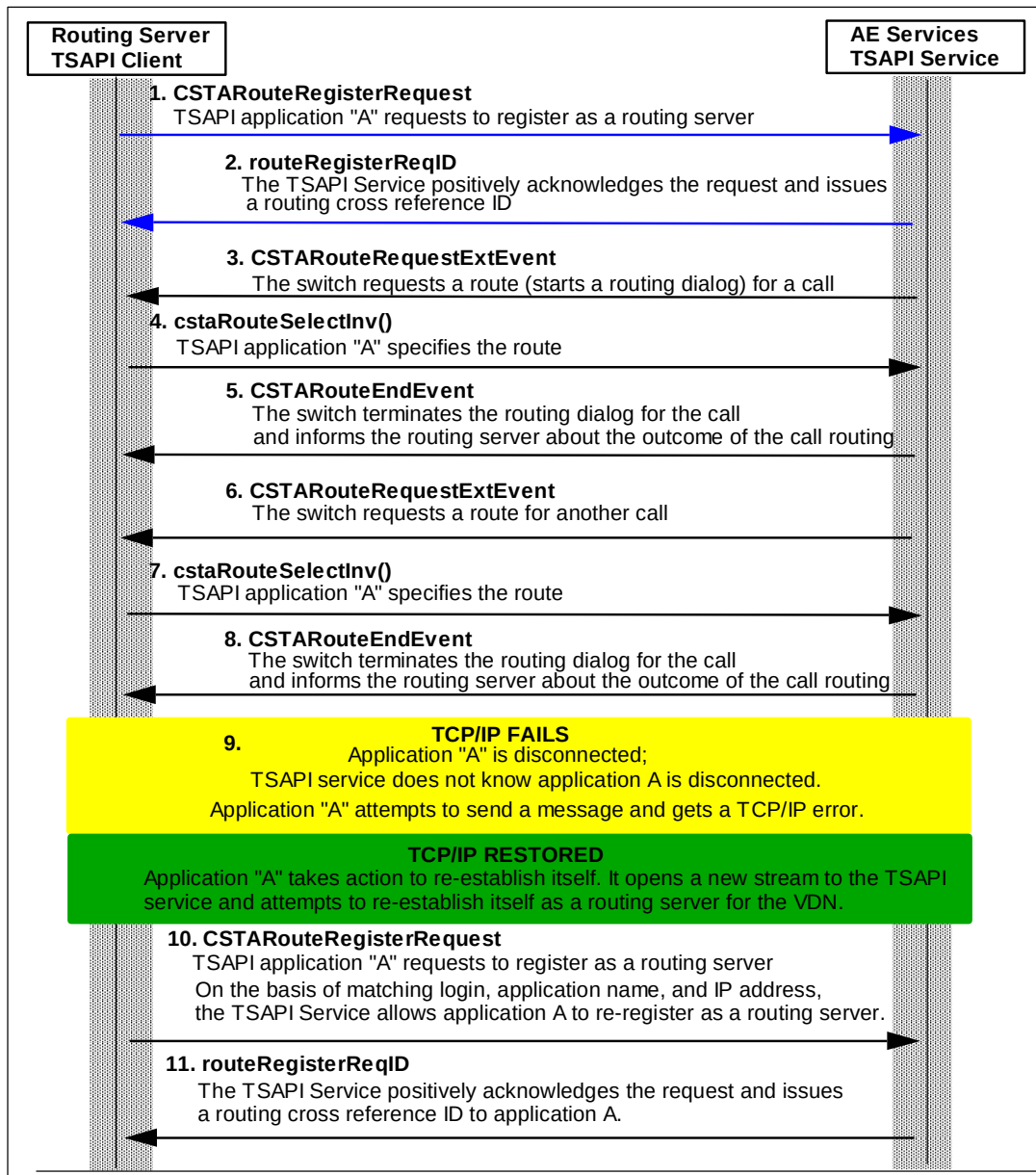


Figure 3: Routing scenario demonstrating TCP/IP failure and recovery



Server-side programming considerations

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

- [Multiple AE Services server considerations](#) on page 53
- [CTI Link Availability](#) on page 54

Multiple AE Services 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 number of simultaneous `cstaMonitorDevice()` requests on a single station device is limited to four per Communication Manager server. That is, a maximum of four AE Servers can monitor the same station at the same time.
- The number of simultaneous `cstaMonitorDevice()` requests on a single ACD split device is limited to eight per Communication Manager server. That is, a maximum of eight AE Servers can use `cstaMonitorDevice()` to monitor the same ACD split at the same time.
- The number of simultaneous `cstaMonitorCallsViaDevice()` monitor requests on one ACD device (VDN or ACD split) is limited to six per Communication Manager server. That is, a maximum of six AE Servers can use `cstaMonitorCallsViaDevice()` to 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.

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.

NOTE:

Beginning with AE Services 6.1, this behavior can be changed through OAM by setting the TSAPI Service Advertising Mode to “Advertise only those Tlinks that are currently in service”. To administer this setting, log into the AE Services Management Console and select AE Services > TSAPI > TSAPI Properties > Advanced Settings.

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 the link has become available again, the request will succeed. If the link is still unavailable, 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 for any previously registered routing devices. The application could make use of System and Link Status Notification (see [Chapter 13: System Status Service Group](#) on page 961) 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.

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](#) on page 56
- [Opening, Closing and Aborting an ACS stream](#) on page 57
- [Sending CSTA Requests and Responses](#) on page 61
- [Receiving Events](#) on page 62
- [Specifying TSAPI versions when you open a stream](#) on page 64
- [Requesting private data when you open an ACS stream](#) on page 66
- [Querying for Available Services](#) on page 66
- [ACS functions and confirmation events](#) on page 67
- [CSTA control services and confirmation events](#) on page 122

Control Services provided by TSAPI

The TSAPI interface, provides two kinds of control services:

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

Applications use ACS to manage their interactions with the AE Services TSAPI Service. 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, the application may initialize 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 123.
- Query for a list of Devices that CSTA Services can monitor, control or route for on an open ACS stream, see [cstaGetDeviceList\(\)](#) on page 128.
- Query to determine if CSTA Call/Call Monitoring is available on an open ACS stream, see [cstaQueryCallMonitor\(\)](#) on page 132.

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. Each application must open an ACS stream before it requests any services.

The TSAPI service can be set up to do security checking to ensure that an application receives CSTA services only for permitted devices.

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 to the TSAPI Service for a particular advertised service. 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 Aura® Application Enablement Services TSAPI 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 cannot 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 `invokeIDs` 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 `invokeIDs` 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 `invokeIDs`. 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 Windows 7, 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.

A TSAPI Windows 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. See [acsOpenStream\(\)](#) on page 68.

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

TS n - n : n

where:

TS is a fixed string value (use uppercase characters) introducing the list of requested TSAPI versions.

n is a number indicating a requested TSAPI version

: the (colon) character is a separator for list entries.

- the (hyphen) character indicates that a list entry represents a range of versions.

Example

The following examples are equivalent and illustrate how an application specifies that it can use TSAPI versions 1 and 2:

TS1-2

TS1 : 2

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 1025).

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 203.
- 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 205.

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\(\)](#) on page 68
- [ACSOpenStreamConfEvent](#) on page 75
- [acsCloseStream\(\)](#) on page 77
- [ACSCloseStreamConfEvent](#) on page 79
- [ACSUniversalFailureConfEvent](#) on page 81
- [acsAbortStream\(\)](#) on page 83
- [acsGetEventBlock\(\)](#) on page 84
- [acsGetEventPoll\(\)](#) on page 87
- [acsGetFile\(\) \(Linux\)](#) on page 90
- [acsSetESR\(\) \(Windows\)](#) on page 91
- [acsEventNotify\(\) \(Windows\)](#) on page 93
- [acsFlushEventQueue\(\)](#) on page 96
- [acsEnumServerNames\(\)](#) on page 98
- [acsGetServerID\(\)](#) on page 100
- [acsQueryAuthInfo\(\)](#) on page 101
- [acsSetHeartbeatInterval\(\)](#) on page 104
- [ACSSetHeartbeatIntervalConfEvent](#) on page 106
- [acsErrorString\(\)](#) on page 108
- [acsReturnCodeString\(\)](#) on page 109
- [acsReturnCodeVerboseString\(\)](#) on page 110
- [ACS Unsolicited Events](#) on page 111
- [ACS Data Types](#) on page 115
- [CSTA control services and confirmation events](#) on page 122
- [CSTA Event Data Types](#) on page 136

acsOpenStream()

An application uses `acsOpenStream()` to open an ACS stream to an advertised service, also known as a Tlink. Generally speaking, 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. (One exception to this rule is `acsEnumServerNames()`.) The function `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 allows the TSAPI client library to select an alternate Tlink if the Tlink specified in the `acsOpenStream()` request is not available when the procedure is executed. To enable 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 Aura® Application Enablement Services TSAPI and CVLAN Client and SDK Installation Guide*, 02-300543.

Syntax

```
#include <acs.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

<code>acsHandle</code>	The <code>acsOpenStream()</code> 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 <code>acsOpenStream()</code> is successful, the application must use this <code>acsHandle</code> in all other function calls to TSAPI on this stream. If <code>acsOpenStream()</code> is successful, TSAPI guarantees that the application has a valid <code>acsHandle</code>. If <code>acsOpenStream()</code> is not successful, then the function return code gives the cause of the failure.
<code>invokeIDType</code>	The application sets the type of invoke identifiers used on the stream being opened. The possible types are as follows: • Application-Generated <code>invokeIDs</code> (<code>APP_GEN_ID</code>) When <code>APP_GEN_ID</code> is selected, the application will provide an <code>invokeID</code> with every TSAPI function call that requires an <code>invokeID</code>. TSAPI will return the supplied <code>invokeID</code> value to the application in the confirmation event for the service request. Application-generated <code>invokeID</code> values can be any 32-bit value. • Library generated <code>invokeIDs</code> (<code>LIB_GEN_ID</code>) When <code>LIB_GEN_ID</code> is selected, the ACS Library will automatically generate an <code>invokeID</code> and will return its value upon successful completion of the function call. The value will be the return from the function call (<code>RetCode_t</code>). Library-generated <code>invokeIDs</code> are always in the range 1 to 32767.
<code>invokeID</code>	The application supplies this handle for matching the <code>acsOpenStream()</code> service request with its confirmation event. An application supplies a value for <code>invokeID</code> only when the <code>invokeIDType</code> parameter is set to <code>APP_GEN_ID</code>. TSAPI ignores the <code>invokeID</code> parameter when <code>invokeIDType</code> parameter is set to <code>LIB_GEN_ID</code>.
<code>streamType</code>	The application provides the type of stream in <code>streamType</code>. The possible values are: • <code>ST_CSTA</code> – 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 <code>csta</code> or <code>CSTA</code>. • <code>ST_OAM</code> – requests an OAM stream. (The AE Services TSAPI Service does not support this value).

<code>serverID</code>	The application provides a null-terminated string of maximum size <code>ACS_MAX_SERVICEID</code>. This string contains the name of an advertised service (in ASCII format). Note: When the TSAPI client configuration file specifies alternate server IDs (Tlinks) for this <code>serverID</code>, the stream may be opened to one of the alternate server IDs instead of the requested server ID. Beginning with AE Services 4.1, an application that requires the actual <code>serverID</code> for the stream can call <code>acsGetServerID()</code>.
<code>loginID</code>	The application provides a pointer to a null terminated string of maximum size <code>ACS_MAX_LOGINID</code>. This string contains the login ID of the user requesting access to the advertised service given in the <code>serviceID</code> parameter. Note: When the TSAPI client configuration file specifies alternate server IDs (Tlinks) for the <code>serverID</code>, the <code>loginID</code> and <code>passwd</code> specified by the application in the <code>acsOpenStream()</code> request should be configured identically for each AE Server.
<code>Passwd</code>	The application provides a pointer to a null terminated string of maximum size <code>ACS_MAX_PASSWORD</code>. This string contains the password of the user given <code>loginID</code>. Note: When the TSAPI client configuration file specifies alternate server IDs (Tlinks) for the <code>serverID</code>, the <code>loginID</code> and <code>passwd</code> specified by the application in the <code>acsOpenStream()</code> request should be configured identically for each AE Server.
<code>applicationName</code>	The application provides a pointer to a null terminated string of maximum size <code>ACS_MAX_APPNAME</code>. This string contains an application name. The system uses the application name on certain administration and maintenance status displays.
<code>acsLevelReq</code>	This version of TSAPI ignores this parameter.
<code>apiVer</code>	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 64.

<code>sendQSize</code>	The application specifies in <code>sendQSize</code> the maximum number of outgoing messages the TSAPI client library will queue before returning <code>ACSERR_QUEUE_FULL</code>. If the application supplies a zero (0) value, then a default queue size will be used. Beginning with Release 6.1 of the TSAPI Windows client library, the default value is 4. Prior to Release 6.1, the default value was 2. Note: The Linux TSAPI client library does not use the <code>sendQSize</code> parameter.
<code>sendExtraBufs</code>	The application specifies the number of additional packet buffers TSAPI allocates for the send queue. If <code>sendExtraBufs</code> 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 <code>ACSERR_NOBUFFERS</code>, it indicates that the application has not allocated enough buffers. Note: The Linux TSAPI client library does not use the <code>sendExtraBufs</code> parameter.
<code>recvQSize</code>	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 <code>recvQSize</code> is set to zero (0). Beginning with Release 6.1 of the TSAPI Windows client library, the default value is 16. Prior to Release 6.1, the default value was 2. Note: The Linux TSAPI client library does not use the <code>recvQSize</code> parameter.
<code>recvExtraBufs</code>	The application specifies the number of additional packet buffers that TSAPI allocates for the receive queue. If <code>recvExtraBufs</code> 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 <code>ACSERR_STREAM_FAILED</code>, then the application does not use <code>recvExtraBufs</code> to allocate enough buffers. Note: The Linux TSAPI client library does not use the <code>recvExtraBufs</code> 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 203.

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 (<0) error condition will be returned. For library-generated identifiers the return value 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 (<0) error 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 on the network.
 - `ACSERR_NORESOURCE` – There are insufficient resources to open an 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 with valid key usage or extended key usage values or that the server certificate is either not provided or not signed by a trusted Certificate Authority.
 - `ACSERR_SSL_FQDN_MISMATCH` – This return value indicates that a stream could not be opened because the fully qualified domain name (FQDN) in the server certificate does not match the expected FQDN.
 - `ACSERR_STREAM_FAILED` – the application attempted 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 an `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 for each AE Services server that appears in the TSAPI client configuration file 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`, `CSTAUniversalFailureConfEvent` or an `ACSUniversalFailureEvent` 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 100.

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

<code>acsHandle</code>	This is the handle for the ACS stream.
<code>eventClass</code>	This is a tag with the value <code>ACSCONFIRMATION</code> , which identifies this message as an ACS confirmation event.
<code>eventType</code>	This is a tag with the value <code>ACS_OPEN_STREAM_CONF</code> , which identifies this message as an <code>ACSOpenStreamConfEvent</code> .
<code>invokeID</code>	This parameter specifies the requested instance of the function or event. It is used to match a specific function request with its confirmation event.
<code>apiVer</code>	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 <code>apiVer</code> 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 64.
<code>libVer</code>	This parameter indicates which version of the Library is running.
<code>tsrvVer</code>	This parameter indicates which version of the TSAPI Service is running.
<code>drvvrVer</code>	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 TSAPI Service. The application may now request call control services from the TSAPI Service 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 TSAPI Service.

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 will contain 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 100.

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()` 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 API Client Library and can be used as an escape mechanism to provide implementation specific commands between the application and the TSAPI Service.

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 (<0) error 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 (<0) error 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 TSAPI Service has closed and that the associated `acsHandle` is no longer valid. This message contains any Private data returned by the TSAPI Service.

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 processing 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 111.

Syntax

The following structure shows only the relevant portions of the unions for this message. See [ACS Data Types](#) on page 115 and [CSTA Event Data Types](#) on page 136 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 a non-telephony problem (no memory, TSAPI Service Security Database check failed, etc.) is encountered while processing a Telephony request 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 TSAPI Service. The application will be unable to request services from the TSAPI Service 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 TSAPI Service.

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 (< 0) error condition. 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 event buffer. The ACS event is still on the API Library queue.

Alternatively, this return value may indicate that the private data length indicated in the `privateData` parameter was smaller than the size of the private data accompanying the next available event for the application on the ACS stream. The API library does not update the value of the `eventBufSize` variable in this case. The application should call `acsGetEventBlock()` again with a larger private data 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 an `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 (< 0) error condition. 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 event buffer. The ACS event is still on the API Library queue.

Alternatively, this return value may indicate that the private data length indicated in the `privateData` parameter was smaller than the size of the private data accompanying the next available event for the application on the ACS stream. The API library does not update the value of the `eventBufSize` variable in this case. The application should call `acsGetEventPoll()` again with a larger private data 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 TSAPI Service.

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 Linux 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 Linux file descriptor is to be returned.

Return Values

This function returns either a Linux 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 Linux 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 a Windows 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 pascal (*EsrFunc)(unsigned long esrParam)

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

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 (< 0) error condition. 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` passed to 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 Windows 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,          /* INPUT */
    HWND             hwnd,              /* INPUT */
    /*
    UINT             msg,                /* INPUT */
    /*
    Boolean           notifyAll);        /* INPUT */
```

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:

<code>wParam</code>	<code>acsHandle</code>
<code>HIWORD(lParam)</code>	<code>eventClass</code>
<code>LOWORD(lParam)</code>	<code>eventType</code>

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 (< 0) error condition. 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_TSAPI_EVENT 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:
        /* Application initialization. */

        /*
         * Indicate that the TSAPI client library should
         * post a WM_TSAPI_EVENT message to this window whenever
         * a TSAPI event arrives on the specified acsHandle.
         */
        acsEventNotify(acsHandle, hwnd, WM_TSAPI_EVENT, TRUE);

        /* other initialization, etc... */
        return 0;

    case WM_TSAPI_EVENT:
        /*
         * wParam contains an ACSHandle_t
         * HIWORD(lParam) contains an EventClass_t
         * LOWORD(lParam) contains an EventType_t
         * Dispatch the TSAPI event to a user-defined
         * handler function here.
         */
        return 0;

        /* cases for other Windows messages... */
    }

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

acsFlushEventQueue()

This function removes all events for the application on an 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 (< 0) error condition. 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 TSAPI Service.

There is no confirmation event for this function.

Application Notes

The application should exercise caution when calling this function, since all events from the TSAPI Service 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 cannot delete the `ACSCloseStreamConfEvent`, since this function cannot 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 `CSTA-UniversalFailureConfEvent`, 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 synchronous and has no associated confirmation event.

Syntax

```
#include <acs.h>

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

RetCode_t    acsEnumServerNames(
    StreamType_t    streamType,        /* INPUT */
    EnumServerNamesCB    callback,      /* INPUT */
    unsigned long    lParam);          /* INPUT */
```

Parameters

streamType

indicates the type of stream requested. The only supported stream type is `ST_CSTA`.

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 (< 0) error condition. 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 cannot communicate with the TSAPI Service. Perhaps the IP address or hostname of the AE Services server is not configured properly in the TSAPI client configuration file; perhaps there is a network issue; or perhaps the TSAPI Service is not running.

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 on Alternate Tlink entries, see the *Avaya Aura® Application Enablement Services TSAPI 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:

ACSPPOSITIVE_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()`. The AE Services TSAPI Service always sets `authInfo.authType` to this value.

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()

Beginning with AE Services 4.1.0, a heartbeat mechanism allows the TSAPI client library to determine when the TSAPI Service is no longer available.

When an ACS stream is idle, the TSAPI Service sends the client library heartbeat messages at a regular interval. (The default heartbeat interval is 20 seconds.) If the TSAPI client library does not receive any messages from the TSAPI Service within two heartbeat intervals, then the library concludes that the TSAPI Service is no longer available, and closes the ACS stream.

An application may use `acsSetHeartbeatInterval()` to change the heartbeat interval for an individual stream. Valid values for the heartbeat interval are 5 – 60 seconds. The heartbeat mechanism cannot be disabled.

If an invalid heartbeat interval is requested (less than 5 seconds or greater than 60 seconds), then the request is rejected. Otherwise, when the TSAPI Service receives the request, it changes the heartbeat interval for the stream and responds 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. Currently, AE Services ignores the value of this parameter.

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.

If the TSAPI Client library closes an ACS stream because it has not received any events for two heartbeat intervals, it notifies the application with an `ACSUnsolicitedACSUniversalFailureEvent` with error `TSERVER_STREAM_FAILED`.

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;          /* ACSCONFIRMATION */
    EventType_t      eventType;          /*
ACS_SET_HEARTBEAT_INTERVAL_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    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.

acsErrorString()

This function returns a pointer to a string describing an ACSUniversalFailure_t error value. It is analogous to the C standard library strerror() function.

For example, acsErrorString() will return a pointer to the string “Stream failed” for the ACSUniversalFailure_t error value TSERVER_STREAM_FAILED.

Syntax

```
#include <acs.h>

const char *acsErrorString(ACSUniversalFailure_t error);
```

Parameters

error

This is the error value from an [ACSUniversalFailureConfEvent](#) or an [ACSUniversalFailureEvent](#).

Return Values

This function returns an appropriate error description string, or an “Unknown error” message if the ACS Universal Failure error is unknown.

Application Notes

This function is available beginning with AE Services Release 5.2 of the Windows and Linux TSAPI client libraries.

There is no confirmation event for this function.

Use [cstaErrorString\(\)](#) to obtain a string describing an CSTAUniversalFailure_t error value.

acsReturnCodeString()

This function returns a pointer to a string describing a return code value with type `RetCode_t`. (Most TSAPI functions return a value with type `RetCode_t`.)

For example, `acsReturnCodeString()` will return a pointer to the string “Bad parameter” for the `RetCode_t` value `ACSERR_BADPARAMETER`.

Syntax

```
#include <acs.h>

const char *acsReturnCodeString(RetCode_t returnCode);
```

Parameters

returnCode

This is the return code from a TSAPI client library function with return type `RetCode_t`.

Return Values

This function returns an appropriate string to describe the return code, or an “Unknown return code” message if the return code is unknown.

Application Notes

This function is available beginning with AE Services Release 5.2 of the Windows and Linux TSAPI client libraries.

There is no confirmation event for this function.

For a more verbose description of the return code, use function [`acsReturnCodeVerboseString\(\)`](#) instead.

acsReturnCodeVerboseString()

This function returns a pointer to a verbose string describing a return code value with type `RetCode_t`. (Most TSAPI functions return a value with type `RetCode_t`.)

For example, `acsReturnCodeVerboseString()` will return a pointer to the string “One or more of the supplied parameters was invalid” for the `RetCode_t` value `ACSERR_BADPARAMETER`.

Syntax

```
#include <acs.h>

const char *acsReturnCodeVerboseString(RetCode_t returnCode);
```

Parameters

returnCode

This is the return code from a TSAPI client library function with return type `RetCode_t`.

Return Values

This function returns an appropriate string to describe the return code, or the message “The TSAPI client library returned an unrecognized value” if the return code is unknown.

Application Notes

This function is available beginning with AE Services Release 5.2 of the Windows and Linux TSAPI client libraries.

There is no confirmation event for this function.

For a less verbose description of the return code, use function [acsReturnCodeString\(\)](#) instead.

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.

Syntax

The following structure shows only the relevant portions of the unions for this message. See [ACS Data Types](#) on page 115 and [CSTA Event Data Types](#) on page 135 for a complete description of the event structure.

```
typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* ACSUNSOLICITED */
    EventType_t      eventType;          /* ACS_UNIVERSAL_FAILURE */
} 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](#) on page 116
- [ACS Event Data Types](#) on page 121

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

        * resources to open an

        * 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

        * encountered an

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

typedef unsigned short EventType_t;          /* event types are

        * defined in acs.h

        * and csta.h */

/* defines for ACS event types */
#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
#define ACS_SET_HEARTBEAT_INTERVAL        14

```

```
#define ACS_SET_HEARTBEAT_INTERVAL_CONF    15

typedef char Boolean;
typedef char Nulltype;

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 136).

```
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 discussed in this section are:

- [cstaGetAPICaps\(\)](#) on page 123
- [CSTAGetAPICapsConfEvent](#) on page 125
- [cstaGetDeviceList\(\)](#) on page 128
- [CSTAGetDeviceListConfEvent](#) on page 130
- [cstaQueryCallMonitor\(\)](#) on page 132
- [CSTAQueryCallMonitorConfEvent](#) on page 133
- [cstaErrorString\(\)](#) on page 135

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 confirmation event for this service will provide 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 (<0) error 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 (<0) error 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 TSAPI Service 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 `ACSPOSITIVE_ACK`, the request has been forwarded to the TSAPI Service, 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 TSAPI Service 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 136 for a complete description of the event structure.

```
typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTA_CONFIRMATION */
    EventType_t      eventType;          /* CSTA_GETAPI_CAPS_CONF */
} 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 a `CSTAGetAPICapsConfEvent`. For information about the private data associated with the `CSTAGetAPICapsConfEvent` see:

- [CSTA Get API Capabilities confirmation structures for Private Data Version 11 and later](#) on page 206.
- [CSTA Get API Capabilities Private Data Versions 8 – 10 Syntax](#) on page 1032
- [CSTA Get API Capabilities for Private Data Version 7](#) on page 1037
- [CSTAGetAPICaps Confirmation interface structures for Private Data Versions 4, 5, and 6](#) on page 1050

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 for which a user has privileges, so this API call will return only `CSTA_MAX_GET_DEVICE` devices in any one `CSTAGetDeviceListConfEvent`, which means several calls to `cstaGetDeviceList()` may be necessary to retrieve all the devices. The value of `index` should be set of -1 the first time this function 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()` service. 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 (<0) error 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 (<0) error 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 provides 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 115 and [CSTA Event Data Types](#) on page 136 for a complete description of the event structure.

```
typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;      /* CSTACONFIRMATION */
    EventType_t      eventType;      /* CSTA_GET_DEVICE_LIST_CONF */
} ACSEventHeader_t;

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

typedef struct CSTAGetDeviceListConfEvent_t {
    SDBLevel_t      driverSdbLevel;
    CSTALevel_t     level;
    long            index;
    DeviceList      devList;
} CSTAGetDeviceListConfEvent_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;

```

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 a `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` corresponding to the devices for this stream.

cstaQueryCallMonitor()

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

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 (<0) error 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 (<0) error 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 TSAPI Service 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. It indicates whether or not the ACS stream has call/call monitoring privileges in the security database.

Syntax

The following structure shows only the relevant portions of the unions for this message. See [ACS Data Types](#) on page 115 and [CSTA Event Data Types](#) on page 136 for a complete description of the event structure.

```
typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass; /* CSTACONFIRMATION */
    EventType_t      eventType; /* CSTA_QUERY_CALL_MONITOR_CONF
*/
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    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 privileges in the security database.

cstaErrorString()

This function returns a pointer to a string describing a CSTAUniversalFailure_t error value. It is analogous to the C standard library strerror() function.

For example, cstaErrorString() will return a pointer to the string “Invalid CSTA device identifier” for the CSTAUniversalFailure_t error value INVALID_CSTA_DEVICE_IDENTIFIER.

Syntax

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

const char *cstaErrorString(CSTAUniversalFailure_t error);
```

Parameters

error

This is the error value from a [CSTAUniversalFailureConfEvent](#).

Return Values

This function returns an appropriate error description string, or an “Unknown error” message if the CSTA Universal Failure error value is unknown.

Application Notes

This function is available beginning with AE Services Release 5.2 of the Windows and Linux TSAPI client libraries.

There is no confirmation event for this function.

Use [acsErrorString\(\)](#) to obtain a string describing an ACSUniversalFailure_t error value.

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 115. 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 (`CSTA_MAKE_CALL_CONF`, `CSTA_HELD` 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;
        CSTARouteRequest_t               reRouteRequest;
        CSTAEscapeSvcReqEvent_t           escapeSvcReqEvent;
        CSTASysStatReqEvent_t             sysStatRequest;
    } u;
} CSTARouteRequestEvent;

typedef struct
{
    union
    {
        CSTARouteRegisterAbortEvent_t     registerAbort;
        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           transferCall;
    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](#) on page 143
- [CSTA Objects](#) on page 148

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 (private) Clear Call Clear Connection Conference Call Consultation Call Consultation-Direct-Agent Call (private) Consultation Supervisor-Assist Call (private) Deflect Call Hold Call Make Call (Private) 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) Send DTMF Tone (private) Single Step Conference (private) Single Step Transfer Call (private) Transfer Call
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

Table 6: Supported CSTA Services for Communication Manager

Service Group	Service Group Definition	Supported Service(s)
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 State Query Call Classifier (private) Query Device Info Query Device Name Query Do Not Disturb Query Endpoint Registration Info (private) Query Forwarding Query Message Waiting Indicator Query Time of Day (private) Query Trunk Group (private) Query Station Status (private) Query Universal Call ID (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

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) • 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 • Message Waiting Endpoint Registration Event Reports: • Endpoint Registered (private) • Endpoint Unregistered (private)
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

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
Routing	Re-Route Event
Escape	Send Private Event
Maintenance	Back in Service Event Out of Service Event

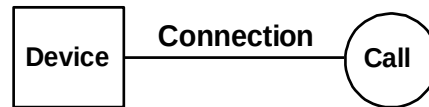
Table 7: Unsupported CSTA Services

Service Group	Unsupported Service(s) or Event Report(s)
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



The CSTA Device object

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 148
- Device Class – for more information, see [Device Class](#) on page 149
- Device Identifier – for more information, see [Device Identifier](#) on page 149

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.

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*.

Table 8: CSTA Device Type Definitions

CSTA Type	Definition	Communication Manager Object
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 link (ASAI), modem pool, etc.

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 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

The TSAPI Service provides dynamic device identifiers under several circumstances.

When an event contains a Connection Identifier and only the `callID` component of that Connection Identifier is relevant, the TSAPI Service sets the `deviceID` component of the Connection Identifier to "0" and the `devIDType` component to `DYNAMIC_ID`. For example, in the [Call Cleared Event](#), the `clearedCall` parameter is a Connection Identifier, but only the `callID` component of this Connection Identifier is meaningful.

When a call is connected through a trunk with an unknown device identifier, a dynamic device identifier, known as a *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.

⇒ NOTE:

In some cases, a trunk identifier may represent a local device rather than an external endpoint. See [Connection Identifier Conflict](#) on page 158 for more detail.

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, as 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. The device identifier associated with this endpoint may be a public number. Prefix or escape digits may be present.
- `EXPLICIT_PUBLIC_UNKNOWN (30)` – There are two cases for this type:
 - 1) 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.
 - 2) 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. 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:
 - 1) 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.

2) 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;

```


Device History

Beginning with private data version 7, 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 drops off a call).

Note:

The device history cannot be guaranteed for events that happened before monitoring started. Note that the cause value should be `EC_NETWORK_SIGNAL` 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

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `DeviceID` that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

Parameter	Description
olddeviceID	[mandatory – supported] 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 <code>divertingDevice</code> 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 may also be: • “Not Known” – indicates that the device identifier cannot be provided. • “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and <code>oldconnectionID</code> are not provided.
cause	[optional – supported] The reason the device left the call or was redirected. This information should be consistent with the <code>cause</code> provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
oldconnectionID	[optional – supported] The CSTA <code>ConnectionID</code> that represents the last <code>ConnectionID</code> 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 <code>ConnectionID</code> provided in the Diverted, Transferred, or Connection Cleared event).

The value of the device history count parameter

For AE Services, the value of device history `count` parameter is at most 1.

The device history `count` parameter indicates the number of entries in the `deviceHistory` parameter. AE Services supports only one entry in the `deviceHistory` parameter. When the limit of one is reached, a new value replaces the old value (unless specifically stated otherwise in this document).

Merging calls – DeviceHistory

When merging calls during a conference or transfer, the source for `DeviceHistory` data is always the Primary Old Call.

Interactions:

Beginning with private data version 7, the private data accompanying a `CSTAGetAPICapsConfEvent` provides a field named `devicehistoryCount`. This field indicates the maximum value of the `deviceHistory count`. For AE Services, this value is always 1 (one).

The CSTA 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 157
- State – see [Call State](#) on page 158

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 provided */
    DeviceID_t                          deviceID;             /* set to "0"
when only

                                * callID is of interest */
```

```

        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 CSTA Connection object

A “connection,” as defined by CSTA, is a relationship that exists between a call and a device. Many TSAPI 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 158.
- State – for more information, see [Connection State](#) on page 160.

Connection Identifier (ConnectionID)

A `ConnectionID` is a combination of a Call Identifier (`callID`) and a Device Identifier (`deviceID`). The `ConnectionID` is unique within a Communication Manager server. An application cannot 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. If the `callID` is the only value that is present, then the `deviceID` is set to “0” and the device identifier type (`devIDType`) is set to `DYNAMIC_ID`.

A `ConnectionID` may also contain a static device identifier (e.g., a dial string or local extension number) or a dynamic device identifier (e.g., a trunk identifier).

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 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 in [Table 9](#).

Figure 5: AE Services TSAPI Service Sample Connection State Model

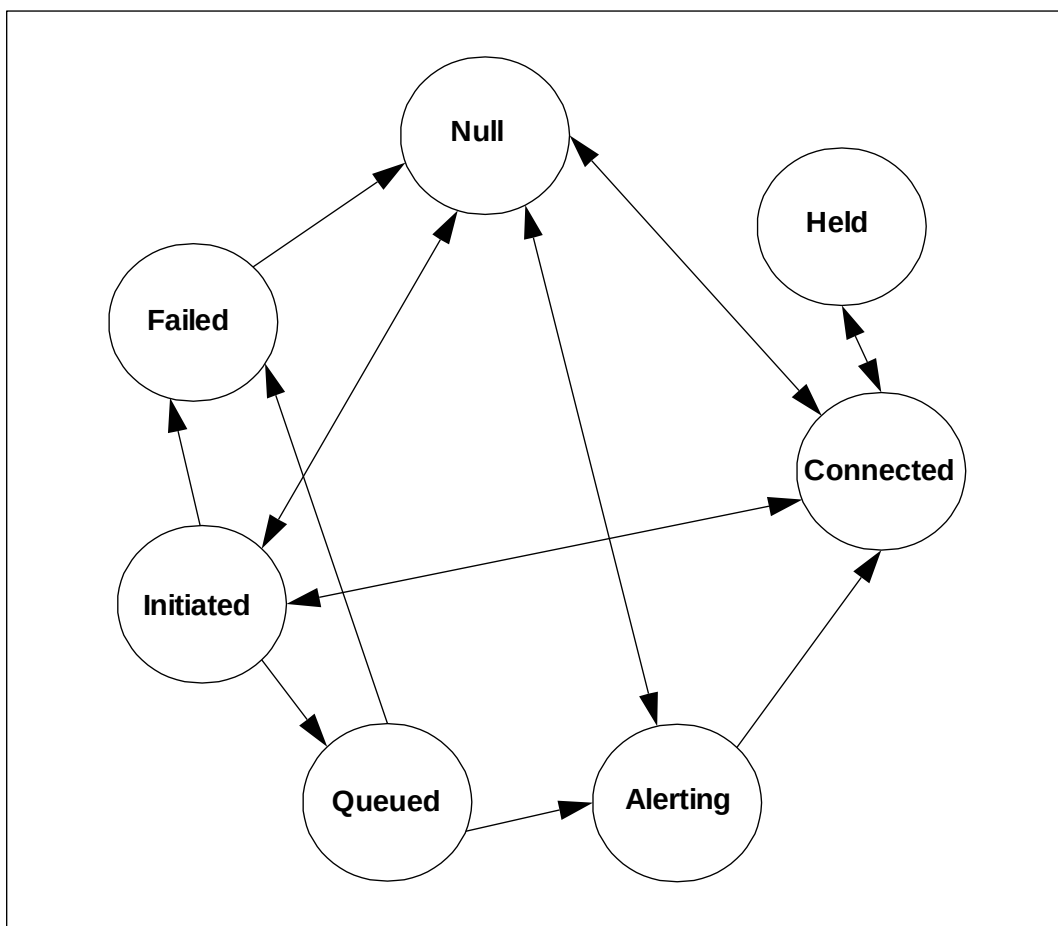


Table 9: TSAPI Service Connection State Definitions

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

Chapter 5: Avaya TSAPI Service Private Data

This chapter describes the private data features provided by the Avaya Aura® Application Enablement Services (AE Services) TSAPI Service. Topics include:

- [What is private data?](#) on page 175
- [What is a private data version?](#) on page 176
- [Linking your application to the private data functions](#) on page 177Private Data Version 17 features
- AE Services Release 8.1.3 provides the following new service observe APIs with Private Data Version 17:

Service Observe Filter

Service Observe State Change Event

Service Observe Activation

Service Observe Deactivation

Service Observe Set Connection Mode

Query Service Observe Status

Service Observe Filter

Starting private data version 17 AES shall support a new private filter as follows:

attServiceObserveFilter Value = 0x08.

The filter shall be defined with the define ATT_SERVICE_OBSERVE_FILTER.

This filter shall be used by the client to indicate to the server that the client needs to be updated about any events related to service observe that happen on the Service Observer. This filter shall be supported on cstaMonitorDevice and cstaChangeMonitorFilter. The events are as below:

- Service Observe Activation on endpoint
- Service Observe De-activation on endpoint
- Service Observe Connection Mode Change

It can be set / reset via monitorDevice() and changeMonitorFilter() APIs

Service Observe state change event

From private data version 17, CSTA Private Event message will start supporting the private data type as ATTSERVICEOBSERVESTATEEVENT_t.

This event shall be sent by CM when the monitored observer device enables or disables Service Observe Feature or changes the service observe connection mode.

AES shall be notified by CM on the association that the CTI client established when it started to monitor the service observer device. This event shall be notified in the cstaPrivateStatusEvent ().

This event shall be represented with the following data structure:

```
typedef struct ATTSERVICEOBSERVESTATEEVENT_t {  
    DeviceID_t      observeeDevice;
```

```

        DeviceID_t          agentID;
        ATTDeviceType_t      observeeType;
        ATTSERVICEObserveConnState_t  observerConnStatus;
        Unsigned short       observeeLocation;
    } ATTSERVICEObserveStateEvent_t;

```

The observee type will depend on the type of the user being observed. The observeeType/associatedObserveeType can be:

```

station (ATT_DT_STATION),
agent (ATT_DT_LOGICAL_AGENT),
VDN (ATT_DT_VDN)

```

Service Observe Activation

Starting private data version 17 onwards the parameters to enable service observe shall be passed in private data ATTSERVICEObserveActivate_t.

```

typedef struct ATTSERVICEObserveActivate_t {
    DeviceID_t          observerDevice;
    DeviceID_t          observeeDevice;
    ATTSERVICEObserveConnState_t  observerConnStatus;
    boolean             nextCall;
    Unsigned short       observeeLocation;
} ATTSERVICEObserveActivate_t;

```

Service Observe Deactivation

Starting private data version 17 onwards the parameters to disable service observe shall be passed in private data ATTSERVICEObserveDeactivate_t. The description of the fields is as follows:

```

typedef struct ATTSERVICEObserveDeactivate_t {
    DeviceID_t  observerDevice;
} ATTSERVICEObserveDeactivate_t;

```

Service Observe Set connection mode

Starting private data version 17 onwards the parameters to set service observe connection mode shall be passed in private data ATTSERVICEObserveSetConnMode_t.

```

typedef struct ATTSERVICEObserveSetConnMode_t{
    DeviceID_t          observerDevice;
    ATTSERVICEObserveConnState_t  observerConnStatus;
} ATTSERVICEObserveSetConnMode_t;

```

Query Service observe status

Starting private data version 17, cstaEscapeService shall have a private data function attQueryServiceObserveInfo. TSAPI service shall map the attQueryServiceObserveInfo function to ASAI Service Observe Status request. attQueryServiceObserveInfo shall have the private data parameters specified in ATTQueryServiceObserveInfo_t.

```

typedef struct ATTQueryServiceObserveInfo_t{
    DeviceID_t          observerDevice;
} ATTQueryServiceObserveInfo_t;

```

Client shall associate the attQueryServiceObserveInfo private data function with a new invoke id that shall be retired when response is received

Nak Parameters:

- universalFailure If any SO 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.
- DUPLICATE_INVOCATION_REJECTION (CS3/7) - Service observe is already active on the observer's device. Service observe can be active on either the station extension or agent login-id
 - PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE (CS3/43) - Service Observing cannot be activated because COR permissions to service observe are not assigned to the Observer Station. This error will also be returned if the COR of object being observed restricts the object from being observed.
 - REQUEST_INCOMPATIBLE_WITH_OBJECT (CS3/63) - Service Observing cannot be activated because SO button is not administered on the Observer
 - INVALID_CSTA_DEVICE_IDENTIFIER (CS0/28) - Service Observee extension/VDN/agent is not administered on switch or is of a type that does not support Service Observe feature. For Example Announcement extensions do not support service observe Or Service Observer Extension/Agent login Id is invalid or not administered on switch or is of a type that does not support Service Observe feature. For Example Announcement extensions do not support service observe.
 - INVALID_ALLOCATION_STATE (CS3/15) - If the Observee or the Observer is an Agent-loginID and the agent is not logged-in
 - GENERIC_STATE_INCOMPATIBILITY (CS3/12) - Observer is a logged-in agent and it is not in Aux-work state
 - VALUE_OUT_OF_RANGE (CS3/100) - Location (VDN case) is not in the range of 1-2000

- `GENERIC_SYSTEM_RESOURCE_AVAILABILITY` (CS3/40) - Number of service observers on the object exceeded. 2 Service Observers can observe an Agent and 999 service observers can observe a VDN.
- `CONFERENCE_MEMBER_LIMIT_EXCEEDED` (CS3/42) - Maximum parties in the connection being observed has exceeded. An observer is a party in the call. A call can have either 6 or 12 parties in a call. If a call has maximum number of parties present then Observer cannot be added as a party in this call.
- `SECURITY_VIOLATION` (CS0/52) - Observee has Data Restriction / Observee has Exclusion Active / Observee has Data Privacy Active on call / Observee Extension cannot be changed after SO has been activated. For example if SO is already active for Agent 1 then SO cannot be activated for Agent 2 without first deactivating SO for agent 1 / Observe VDN not optioned. VDN does not have permission to be observed.
- `INVALID_OBJECT_STATE` (CS0/98) - Extension being busy-verified / Attempt to activate Service observe on an agent when service observe is already active on the physical that the agent is logged in on. / Attempt to activate Service observe on an physical station when service observe is already active on the agent that is logged in on the physical station.
- `INVALID_OBJECT_TYPE` (CS3/80) - Coach connection mode cannot be chosen while activating service observe. Connection mode cannot be changed if connection mode at SO activation was set to no-talk. Next-Call mode cannot be changed after SO has been activated.

Private Data Version 15 features

Channel Type in Originated Event

Beginning with private data version 15, the private data accompanying the CSTA Originated Event includes the Channel Type, if available:

```
typedef struct ATTOrientedEvent_t {
    DeviceID_t      logicalAgent;
    ATTUserToUserInfo_t userInfo;
    ATTConsultMode_t consultMode;
    ATTChannelType_t channelType;
} ATTOrientedEvent_t;
```

VideoCall Type in MakeCall

Beginning with private data version 15, the private data accompanying the CSTA MakeCall includes the VideoCall Type, if available:

```
typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2
} ATTVideoCall_t;

typedef struct ATTMakeCall_t {
    DeviceID_t      destRoute;
    unsigned char   priorityCalling;
    ATTUserToUserInfo_t userInfo;
    ATTVideoCall_t  videoCall;
} ATTMakeCall_t;
```

VideoCall Type in AnswerCall

Beginning with private data version 15, the private data accompanying the CSTA AnswerCall includes the VideoCall Type, if available:

```
typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2
} ATTVideoCall_t;

typedef struct ATTAnswerCall_t {
    ATTVideoCall_t  videoCall;
} ATTAnswerCall_t;
```

VideoCall Type in ConsultationCall

Beginning with private data version 15, the private data accompanying the CSTA ConsultationCall includes the VideoCall Type, if available:

```
typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
```

```

        vidTwoway = 2
    } ATTVideoCall_t;

typedef struct ATTConsultationCall_t {
    DeviceID_t      destRoute;
    unsigned char   priorityCalling;
    ATTUserToUserInfo_t userInfo;
    ATTConsultOptions_t consultOptions;
    ATTVideoCall_t  videoCall;
} ATTConsultationCall_t;

```

VideoCall Type in SingleStepTransferCall

Beginning with private data version 15, the private data accompanying the CSTA SingleStepTransferCall includes the VideoCall Type, if available:

```

typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2
} ATTVideoCall_t;

typedef struct ATTSingleStepTransferCall_t {
    ConnectionID_t  activeCall;
    DeviceID_t      transferredTo;
    ATTVideoCall_t  videoCall;
} ATTSingleStepTransferCall_t;

```

SkillInfoList in QueryAgentLoginResp

Beginning with private data version 15, the private data accompanying the CSTA QueryAgentLoginResponse includes the SkillInfoList, if available:

```

typedef struct ATTQueryAgentLoginResp_t {
    ATTPrivEventCrossRefID_t privEventCrossRefID;
    struct {
        unsigned short  count;
        DeviceID_t      device[10];
    } list;
    struct {
        unsigned short  count;
        ATTAgentSkillInfo_t AgentSkillInfo[10];
    } skillInfolist;
} ATTQueryAgentLoginResp_t;

typedef struct ATTAgentSkillInfo_t {
    DeviceID_t      device;
    AgentID_t       agentID;
    ATTSkillLevel_t skillLevel;
    unsigned char   percentAllocation;
} ATTAgentSkillInfo_t;

typedef enum ATTSkillLevelType_t {
    skLvlTypeNotProvided = 0,

```



```

        skLvlTypeRegular = 1,
        skLvlTypeReserved = 2
    } ATTSkillLevelType_t;

typedef struct ATTSkillLevel_t {
    ATTSkillLevelType_t type;
    long                level;
} ATTSkillLevel_t;

```

SkillInfoList in LoggedOnEvent

Beginning with private data version 15, the private data accompanying the CSTA LoggedOnEvent includes the SkillInfoList, if available:

```

typedef struct ATTLoggedOnEvent_t {
    ATTWorkMode_t    workMode;
    ATTSkillLevel_t  skillLevel;
    unsigned char    percentAllocation;
} ATTLoggedOnEvent_t;

typedef enum ATTSkillLevelType_t {
    skLvlTypeNotProvided = 0,
    skLvlTypeRegular = 1,
    skLvlTypeReserved = 2
} ATTSkillLevelType_t;

typedef struct ATTSkillLevel_t {
    ATTSkillLevelType_t type;
    long                level;
} ATTSkillLevel_t;

```

Reserved Data Type in RouteRequestEvent

Beginning with private data version 15, the private data accompanying the CSTA RouteRequestEvent includes the Reserved Data type, if available:

```

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;
    Reserved16_t       data26;
    Reserved31_t       data38;
} ATTRouteRequestEvent_t;

```

Reserved Data Type in RouteSelect

Beginning with private data version 15, the private data accompanying the CSTA RouteSelect includes the Reserved Data type, if available:

```
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;
    DeviceID_t      data17;
    Reserved7_t      data27;
    Reserved31_t      data38;
} ATTRouteSelect_t;
```

Private Data Version 13 features

Channel Type in Delivered Event

Beginning with private data version 13, the private data accompanying the CSTA Delivered event includes the Channel Type, if available:

```
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;
    ATTChannelType_t channelType;
} ATTDeliveredEvent_t;
```

Channel Type in Established Event

Beginning with private data version 13, the private data accompanying the CSTA Established event includes the Channel Type, if available:

```
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;
    ATTChannelType_t channelType;
} ATTEstablishedEvent_t;
```

- Private Data Version 12 features on page 182
- [Private Data Version 11 features](#) on page 192
- [Private Data Version 10 Features](#) on page 197
- [Private Data Version 9 Features](#) on page 200
- [Private Data Version 8 Features](#) on page 202

- [Requesting private data](#) on page 203
- [CSTA Get API Capabilities confirmation structures for Private Data Version 11 and later](#) on page 206
- [Private Data Service sample code](#) on page 208
- [Upgrading and maintaining applications that use private data](#) on page 217
- [Using the private data header files](#) on page 219

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 prefix 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 17 or PDV 17). With the latest product release of the Application Enablement Services (AE Services 8.1.3) an application may ask the TSAPI service to provide data defined for private data versions 2 through 17. Newer features and content are provided with higher numbered private data versions. Private data versioning is inclusive. If you negotiate private data version 17, 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 15 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 15. If the application, in this example, needed to ensure that confirmation event was returned in a format no later than private data version 4, then it should have initially negotiated private data version 4 for the stream, not version 15.

See [Table 10](#) for a summary history of private data versions.

Table 10: History 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
Application Enablement Services 6.1	PDV 2 through 9
Application Enablement Services 6.2	PDV 2 through 10
Application Enablement Services 6.3.0	PDV 2 through 10
Application Enablement Services 6.3.1	PDV 2 through 11
Application Enablement Services 6.3.3	PDV 2 through 12
Application Enablement Services 7.1.1	PDV 2 through 13
Application Enablement Services 7.1.3.1	PDV 2 through 14
Application Enablement Services 8.1	PDV 2 through 15
Application Enablement Services 8.1.2	PDV 2 through 16
Application Enablement Services 8.1.3	PDV 2 through 17

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 clients, this file is `ATTPRIV32.DLL` (for 32 bits arch.) or `ATTPRIV64.DLL` (for 64 bits arch.). For Linux 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 17 in the right column are new features for Release 8.1.3 of the TSAPI Service. For more information previous version of private data, see [Appendix B: Summary of Private data support](#) on page 999.

Table 11: Private Data Summary

Private Data Feature	Initial Private Data Version
Reserved Data type for Avaya Proprietary APIs	PDV 17
Reserved Data type for Avaya Proprietary APIs	PDV 16
Channel Type in Originated Event and VideoCall Type in MakeCall, AnswerCall, ConsultationCall and SingleStepTransfer	PDV 15
Reserved data type added in routeRequestEvent and RouteSelect, SkillInfolist added in LoggedOnEvent and QueryAgentLoginResp	PDV15
Reserved Data type for Avaya Proprietary APIs	PDV 14
Channel Type in Delivered and Established Event	PDV 13
Calling Device in Diverted Event	PDV 12
Endpoint Registration Info Query	PDV 11
Endpoint Registered Event	PDV 11
Endpoint Unregistered Event	PDV 11
Device ID changes in Failed Event	PDV 11
Agent Work Mode in Logged On Event	PDV 10
Consultation Call support for Consult Options	PDV 10
Enhanced Station Status Query	PDV 10
Consult Mode in Held, Service Initiated, and Originated Events	PDV 9
UCID in Single Step Transfer Call Confirmation event	PDV 9
Single Step Transfer Call	PDV 8
Calling Device in Failed Event	PDV 8

Table 11: Private Data Summary

Private Data Feature	Initial Private Data Version
Reserved Data type for Avaya Proprietary APIs	PDV 17
Reserved Data type for Avaya Proprietary APIs	PDV 16
Enhanced Monitor Calls via Device	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 <code>DeviceIDType_t</code> 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
Selective Listening Hold	PDV 5
Selective Listening Retrieve	PDV 5
Set Advice of Charge	PDV 5
Charge Advice Event	PDV 5

Table 11: Private Data Summary

Private Data Feature	Initial Private Data Version
Reserved Data type for Avaya Proprietary APIs	PDV 17
Reserved Data type for Avaya Proprietary APIs	PDV 16
Reason Code in Set Agent State, Query Agent State, and Logged Off 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

Table 11: Private Data Summary

Private Data Feature	Initial Private Data Version
Reserved Data type for Avaya Proprietary APIs	PDV 17
Reserved Data type for Avaya Proprietary APIs	PDV 16
Application Integration with Expert Agent Selection	PDV 2
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 17 features

AE Services Release 8.1.3 provides the following new service observe APIs with Private Data Version 17:

[Service Observe Filter](#)

[Service Observe State Change Event](#)

[Service Observe Activation](#)

[Service Observe Deactivation](#)

[Service Observe Set Connection Mode](#)

[Query Service Observe Status](#)

Service Observe Filter

Starting private data version 17 AES shall support a new private filter as follows:

attServiceObserveFilter Value = 0x08.

The filter shall be defined with the define ATT_SERVICE_OBSERVE_FILTER.

This filter shall be used by the client to indicate to the server that the client needs to be updated about any events related to service observe that happen on the Service Observer. This filter shall be supported on cstaMonitorDevice and cstaChangeMonitorFilter. The events are as below:

- Service Observe Activation on endpoint
- Service Observe De-activation on endpoint
- Service Observe Connection Mode Change

It can be set / reset via monitorDevice() and changeMonitorFilter() APIs

Service Observe state change event

From private data version 17, CSTA Private Event message will start supporting the private data type as ATTSERVICEOBSERVESTATEEVENT_t.

This event shall be sent by CM when the monitored observer device enables or disables Service Observe Feature or changes the service observe connection mode.

AES shall be notified by CM on the association that the CTI client established when it started to monitor the service observer device. This event shall be notified in the cstaPrivateStatusEvent ().

This event shall be represented with the following data structure:

```
typedef struct ATTSERVICEOBSERVESTATEEVENT_t {
    DeviceID_t          observeeDevice;
    DeviceID_t          agentID;
    ATTDeviceType_t      observeeType;
    ATTSERVICEOBSERVECONNSTATE_t  observerConnStatus;
    Unsigned short       observeeLocation;
} ATTSERVICEOBSERVESTATEEVENT_t;
```

The observee type will depend on the type of the user being observed. The observeeType/associatedObserveeType can be:

```
station (ATT_DT_STATION),
agent (ATT_DT_LOGICAL_AGENT),
VDN (ATT_DT_VDN)
```

Service Observe Activation

Starting private data version 17 onwards the parameters to enable service observe shall be passed in private data ATTSERVICEOBSERVEACTIVATE_t.

```
typedef struct ATTSERVICEOBSERVEACTIVATE_t {
    DeviceID_t          observerDevice;
    DeviceID_t          observeeDevice;
    ATTSERVICEOBSERVECONNSTATE_t  observerConnStatus;
    boolean             nextCall;
    Unsigned short      observeeLocation;
} ATTSERVICEOBSERVEACTIVATE_t;
```

Service Observe Deactivation

Starting private data version 17 onwards the parameters to disable service observe shall be passed in private data ATTSERVICEOBSERVEDEACTIVATE_t. The description of the fields is as follows:

```
typedef struct ATTSERVICEOBSERVEDEACTIVATE_t {
    DeviceID_t  observerDevice;
} ATTSERVICEOBSERVEDEACTIVATE_t;
```

Service Observe Set connection mode

Starting private data version 17 onwards the parameters to set service observe connection mode shall be passed in private data ATTSERVICEOBSERVESETCONNMODE_t.

```
typedef struct ATTSERVICEOBSERVESETCONNMODE_t{
    DeviceID_t          observerDevice;
    ATTSERVICEOBSERVECONNSTATE_t  observerConnStatus;
} ATTSERVICEOBSERVESETCONNMODE_t;
```

Query Service observe status

Starting private data version 17, cstaEscapeService shall have a private data function attQueryServiceObserveInfo. TSAPI service shall map the attQueryServiceObserveInfo function to ASAI Service Observe Status request. attQueryServiceObserveInfo shall have the private data parameters specified in ATTQUERYSERVICEOBSERVEINFO_t.

```
typedef struct ATTQUERYSERVICEOBSERVEINFO_t{
    DeviceID_t          observerDevice;
} ATTQUERYSERVICEOBSERVEINFO_t;
```

Client shall associate the attQueryServiceObserveInfo private data function with a new invoke id that shall be retired when response is received

Nak Parameters:

universalFailure	If any SO 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 21: Common switch-related CSTA Service errors -- universalFailure on page 997. • DUPLICATE_INVOCATION_REJECTION (CS3/7) - Service observe is already active on the observer's device. Service observe can be active on either the station extension or agent login-id • PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE (CS3/43) - Service Observing cannot be activated because COR permissions to service observe are not assigned to the Observer Station. This error will also be returned if the COR of object being observed restricts the object from being observed. • REQUEST_INCOMPATIBLE_WITH_OBJECT (CS3/63) - Service Observing cannot be activated because SO button is not administered on the Observer • INVALID_CSTA_DEVICE_IDENTIFIER (CS0/28) - Service Observee extension/VDN/agent is not administered on switch or is of a type that does not support Service Observe feature. For Example Announcement extensions do not support service observe Or Service Observer Extension/Agent login Id is invalid or not administered on switch or is of a type that does not support Service Observe feature. For Example Announcement extensions do not support service observe. • INVALID_ALLOCATION_STATE (CS3/15) - If the Observee or the Observer is an Agent-loginID and the agent is not logged-in • GENERIC_STATE_INCOMPATIBILITY (CS3/12) - Observer is a logged-in agent and it is not in Aux-work state • VALUE_OUT_OF_RANGE (CS3/100) - Location (VDN case) is not in the range of 1-2000 • GENERIC_SYSTEM_RESOURCE_AVAILABILITY (CS3/40) - Number of service observers on the object exceeded. 2 Service Observers can

observe an Agent and 999 service observers can observe a VDN.

- CONFERENCE_MEMBER_LIMIT_EXCEEDED (CS3/42) - Maximum parties in the connection being observed has exceeded. An observer is a party in the call. A call can have either 6 or 12 parties in a call. If a call has maximum number of parties present then Observer cannot be added as a party in this call.
- SECURITY_VIOLATION (CS0/52) - Observee has Data Restriction / Observee has Exclusion Active / Observee has Data Privacy Active on call / Observee Extension cannot be changed after SO has been activated. For example if SO is already active for Agent 1 then SO cannot be activated for Agent 2 without first deactivating SO for agent 1 / Observe VDN not optioned. VDN does not have permission to be observed.
- INVALID_OBJECT_STATE (CS0/98) - Extension being busy-verified / Attempt to activate Service observe on an agent when service observe is already active on the physical that the agent is logged in on. / Attempt to activate Service observe on an physical station when service observe is already active on the agent that is logged in on the physical station.
- INVALID_OBJECT_TYPE (CS3/80) - Coach connection mode cannot be chosen while activating service observe. Connection mode cannot be changed if connection mode at SO activation was set to no-talk. Next-Call mode cannot be changed after SO has been activated.

Private Data Version 15 features

Channel Type in Originated Event

Beginning with private data version 15, the private data accompanying the CSTA Originated Event includes the Channel Type, if available:

```
typedef struct ATTOrientedEvent_t {
    DeviceID_t      logicalAgent;
    ATTUserToUserInfo_t userInfo;
    ATTConsultMode_t consultMode;
    ATTChannelType_t channelType;
} ATTOrientedEvent_t;
```

VideoCall Type in MakeCall

Beginning with private data version 15, the private data accompanying the CSTA MakeCall includes the VideoCall Type, if available:

```
typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2
} ATTVideoCall_t;

typedef struct ATTMakeCall_t {
    DeviceID_t      destRoute;
    unsigned char   priorityCalling;
    ATTUserToUserInfo_t userInfo;
    ATTVideoCall_t  videoCall;
} ATTMakeCall_t;
```

VideoCall Type in AnswerCall

Beginning with private data version 15, the private data accompanying the CSTA AnswerCall includes the VideoCall Type, if available:

```
typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2
} ATTVideoCall_t;

typedef struct ATTAnswerCall_t {
    ATTVideoCall_t  videoCall;
} ATTAnswerCall_t;
```

VideoCall Type in ConsultationCall

Beginning with private data version 15, the private data accompanying the CSTA ConsultationCall includes the VideoCall Type, if available:

```
typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
```



```

        vidTwoway = 2
    } ATTVideoCall_t;

typedef struct ATTConsultationCall_t {
    DeviceID_t      destRoute;
    unsigned char   priorityCalling;
    ATTUserToUserInfo_t userInfo;
    ATTConsultOptions_t consultOptions;
    ATTVideoCall_t  videoCall;
} ATTConsultationCall_t;

```

VideoCall Type in SingleStepTransferCall

Beginning with private data version 15, the private data accompanying the CSTA SingleStepTransferCall includes the VideoCall Type, if available:

```

typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2
} ATTVideoCall_t;

typedef struct ATTSingleStepTransferCall_t {
    ConnectionID_t  activeCall;
    DeviceID_t      transferredTo;
    ATTVideoCall_t  videoCall;
} ATTSingleStepTransferCall_t;

```

SkillInfoList in QueryAgentLoginResp

Beginning with private data version 15, the private data accompanying the CSTA QueryAgentLoginResponse includes the SkillInfoList, if available:

```

typedef struct ATTQueryAgentLoginResp_t {
    ATTPrivEventCrossRefID_t privEventCrossRefID;
    struct {
        unsigned short  count;
        DeviceID_t      device[10];
    } list;
    struct {
        unsigned short  count;
        ATTAgentSkillInfo_t AgentSkillInfo[10];
    } skillInfolist;
} ATTQueryAgentLoginResp_t;

typedef struct ATTAgentSkillInfo_t {
    DeviceID_t      device;
    AgentID_t       agentID;
    ATTSkillLevel_t skillLevel;
    unsigned char   percentAllocation;
} ATTAgentSkillInfo_t;

typedef enum ATTSkillLevelType_t {
    skLvlTypeNotProvided = 0,

```

```

        skLvlTypeRegular = 1,
        skLvlTypeReserved = 2
    } ATTSkillLevelType_t;

typedef struct ATTSkillLevel_t {
    ATTSkillLevelType_t type;
    long                level;
} ATTSkillLevel_t;

```

SkillInfoList in LoggedOnEvent

Beginning with private data version 15, the private data accompanying the CSTA LoggedOnEvent includes the SkillInfoList, if available:

```

typedef struct ATTLoggedOnEvent_t {
    ATTWorkMode_t    workMode;
    ATTSkillLevel_t  skillLevel;
    unsigned char    percentAllocation;
} ATTLoggedOnEvent_t;

typedef enum ATTSkillLevelType_t {
    skLvlTypeNotProvided = 0,
    skLvlTypeRegular = 1,
    skLvlTypeReserved = 2
} ATTSkillLevelType_t;

typedef struct ATTSkillLevel_t {
    ATTSkillLevelType_t type;
    long                level;
} ATTSkillLevel_t;

```

Reserved Data Type in RouteRequestEvent

Beginning with private data version 15, the private data accompanying the CSTA RouteRequestEvent includes the Reserved Data type, if available:

```

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;
    Reserved16_t       data26;
    Reserved31_t       data38;
} ATTRouteRequestEvent_t;

```

Reserved Data Type in RouteSelect

Beginning with private data version 15, the private data accompanying the CSTA RouteSelect includes the Reserved Data type, if available:

```
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;
    DeviceID_t      data17;
    Reserved7_t      data27;
    Reserved31_t      data38;
} ATTRouteSelect_t;
```

Private Data Version 13 features

Channel Type in Delivered Event

Beginning with private data version 13, the private data accompanying the CSTA Delivered event includes the Channel Type, if available:

```
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;
    ATTChannelType_t channelType;
} ATTDeliveredEvent_t;
```

Channel Type in Established Event

Beginning with private data version 13, the private data accompanying the CSTA Established event includes the Channel Type, if available:

```
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;
    ATTChannelType_t channelType;
} ATTEstablishedEvent_t;
```

Private Data Version 12 features

Calling Device in Diverted Event

Beginning with private data version 12, the private data accompanying the CSTA Diverted event includes the Calling Device, if available:

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

Private Data Version 11 features

AE Services Release 6.3.1 provides the following new features with Private Data Version 11:

- [Endpoint Registration Info Query](#)
- [Endpoint Registered Event](#)
- [Endpoint Unregistered Event](#)
- [Failed Event Device Identifiers](#)

Endpoint Registration Info Query

Private data version 11 supports a new escape service, `attQueryEndpointRegistrationInfo()`. This query identifies the H.323 and SIP endpoints registered to a station extension, and also provides the station's service state.

```
attQueryEndpointRegistrationInfo(ATTPrivateData_t *privateData,
                                DeviceID_t *device);
```

The `attQueryEndpointRegistrationInfo()` escape service is supported for ASAI link version 6 (or later). ASAI link version 6 is available beginning with Avaya Communication Manager Release 6.3.2.

The confirmation event for this escape service includes:

- the device ID of the station extension
- the service state of the station extension
- the number of H.323 and/or SIP endpoints registered to the station extension (0-4)

For each registered H.323 and SIP endpoint, the confirmation event also includes:

- an instance ID (0-2)
- the endpoint's IP address (for H.323 endpoints only)
- the endpoint's switch-end IP address
- the endpoint's MAC address
- the endpoint's product ID
- the endpoint's network region (1-250)
- the endpoint's dependency mode (main, independent, or dependent)
- the endpoint's media mode (no media, client/server mode, or telecommuter mode)
- the supported Unicode script character sets
- the endpoint's set type
- the endpoint's signaling protocol (H.323 or SIP)

```
typedef struct ATTQueryEndpointRegistrationInfoConfEvent_t {
```

```

        DeviceID_t                                device;
        ATTSERVICESTATE_t                          serviceState;
        ATTTREGISTEREDENDPOINTLIST_t               registeredEndpoints;
} ATTTQUERYENDPOINTREGISTRATIONINFOCONFEVENT_t;

typedef enum ATTSERVICESTATE_t {
    SS_UNKNOWN = -1,
    SS_OUT_OF_SERVICE = 0,
    SS_IN_SERVICE = 1
} ATTSERVICESTATE_t;

typedef struct ATTTREGISTEREDENDPOINTLIST_t {
    unsigned int                                     count;
    ATTTREGISTEREDENDPOINTINFO_t *endpoint;
} ATTTREGISTEREDENDPOINTLIST_t;

typedef struct ATTTREGISTEREDENDPOINTINFO_t {
    ATTTENDPOINTINSTANCEID_t                       instanceID;
    ATTTENDPOINTADDRESS_t                          endpointAddress;
    ATTTIPADDRESS_t                                switchEndIpAddress;
    ATTTMACADDRESS_t                               macAddress;
    ATTTPRODUCTID_t                                productID;
    short                                           networkRegion;
    ATTTMEDIAMODE_t                                mediaMode;
    ATTTDEPENDENCYMODE_t                           dependencyMode;
    ATTTUNICODESCRIPT_t                            unicodeScript;
    ATTTSTATIONTYPE_t                              stationType;
    ATTTSIGNALINGPROTOCOL_t                        signalingProtocol;
} ATTTREGISTEREDENDPOINTINFO_t;

typedef unsigned short                             ATTTENDPOINTINSTANCEID_t;

typedef char                                       ATTTENDPOINTADDRESS_t[256];

typedef char                                       ATTTIPADDRESS_t[46];

typedef char                                       ATTTMACADDRESS_t[18];

typedef char                                       ATTTPRODUCTID_t[16];

typedef enum ATTTMEDIAMODE_t {
    MM_OTHER = -1,
    MM_NONE = 0,
    MM_CLIENT_SERVER = 1,
    MM_TELECOMMUTER = 2
} ATTTMEDIAMODE_t;

typedef enum ATTTDEPENDENCYMODE_t {
    DM_OTHER = -1,

```

```

        DM_MAIN = 1,
        DM_DEPENDENT = 2,
        DM_INDEPENDENT = 3
    } ATTDependencyMode_t;

typedef unsigned int                ATTUnicodeScript_t;

typedef char                        ATTStationType_t[16];

typedef enum ATTSignalingProtocol_t {
    SP_NOT_PROVIDED = -1,
    SP_H323 = 1,
    SP_SIP = 2
} ATTSignalingProtocol_t;

```

Endpoint Registered Event

Beginning with private data version 11, a device monitor for a station extension will receive a CSTA Private Status event whenever an H.323 or SIP endpoint registers to the station extension. The private data accompanying the event is an `ATTEndpointRegisteredEvent_t`.

The Endpoint Registered Event is supported for ASAI link version 6 (or later). ASAI link version 6 is available beginning with Avaya Communication Manager Release 6.3.2.

```

typedef struct ATTEndpointRegisteredEvent_t {
    DeviceID_t                device;
    ATTServiceState_t         serviceState;
    ATTRegisteredEndpointInfo_t endpointInfo;
} ATTEndpointRegisteredEvent_t;

```

Endpoint Unregistered Event

Beginning with private data version 11, a device monitor for a station extension will receive a CSTA Private Status event whenever an H.323 or SIP endpoint unregisters from the station extension. The private data accompanying the event is an `ATTEndpointUnregisteredEvent_t`.

The Endpoint Unregistered Event is supported for ASAI link version 6 (or later). ASAI link version 6 is available beginning with Avaya Communication Manager Release 6.3.2.

```

typedef struct ATTEndpointUnregisteredEvent_t {
    DeviceID_t                device;
    ATTServiceState_t         serviceState;
    ATTEndpointInstanceID_t instanceID;
    ATTEndpointAddress_t      endpointAddress;
    ATTIPAddress_t            switchEndIpAddress;
    ATTDependencyMode_t       dependencyMode;
    ATTStationType_t          stationType;
    ATTSignalingProtocol_t    signalingProtocol;
    ATTUnregisterReason_t     reason;
}

```



```

        long                                     cmReason;
    } ATTEndpointUnregisteredEvent_t;

typedef enum ATTUnregisterReason_t {
    UR_OTHER = -1,
    UR_LOGOFF = 1,
    UR_NO_SIGNALING_CONNECTION = 2,
    UR_FORCE_LOGIN = 3,
    UR_SIGNALING_CONNECTION_CLOSED = 4,
    UR_Q931_TIMEOUT = 5,
    UR_TTL_EXPIRED = 6,
    UR_EXTENSION_REMOVED = 7,
    UR_SOFTPHONE = 8,
    UR_NO_LAN_PORTS = 9,
    UR_RO_TTI_MTC = 10,
    UR_SYSTEM_RESTART = 11,
    UR_SAT_COMMAND = 12,
    UR_CALL_EXPIRED = 13,
    UR_SHARED_CONTROL = 14,
    UR_TSA = 15,
    UR_DCP_STN_STATE = 16,
    UR_SHARED_CONTROL_MTC = 17,
    UR_MODULE_ID_MISMATCH = 18,
    UR_REGISTRATION_EXPIRED = 19,
    UR_INVALID_STATE = 20,
    UR_AUDIT_FAILURE = 21,
    UR_REMOTE_MAX_MTC = 22,
    UR_NO_H323_CONNECTION = 23,
    UR_LOST_SIGNALING_CONNECTION = 24,
    UR_STATION_MTC = 25
} ATTUnregisterReason_t;

```

Failed Event Device Identifiers

Beginning with private data version 11, device identifiers reported in the CSTA [Failed Event](#) and ATT Failed event are set differently than for earlier private data versions.

Previously, in most scenarios the TSAPI Service would set the following event parameters using the deviceID of the `calledDevice`:

- the `failedConnection deviceID`
- the `failingDevice deviceID`
- the device history `olddeviceID`
- the device history `oldconnectionID device ID`

However, for some call scenarios, this is incorrect.

Because the TSAPI Service cannot always provide the correct device ID in these event parameters, beginning with private data 11 the TSAPI Service sets these parameters differently for some call scenarios. In those scenarios:

- the `failedConnection deviceId` is set to the empty string (“”)
- the `failingDevice deviceId` is set to the empty string (“”) and its device ID status is reported as `ID_NOT_KNOWN`
- the device history `olddeviceId` is set to “Not Known”
- the device history `oldconnectionID deviceId` is set to “Not Known”.

Private Data Version 10 Features

AE Services Release 6.2 provides the following new features with Private Data Version 10:

- [Agent Work Mode in Logged On Event](#)
- [Consultation Call Support for Consult Options](#)
- [Enhanced Station Status Query](#)

Agent Work Mode in Logged On Event

Beginning with private data version 10, private data accompanying a CSTA [Logged On Event](#) provides the initial work mode of the agent. The possible work modes are:

- Auxiliary Work Mode (WM_AUX_WORK)
- After-Call Work Mode (WM_AFTCAL_WK)
- Auto-In Mode (WM_AUTO_IN)
- Manual-In Mode (WM_MANUAL_IN)

Consultation Call Support for Consult Options

Applications may use the new private data formatting function `attV10ConsultationCall()` to express the intended purpose of a `cstaConsultationCall()` service request. This function is similar to the pre-existing function `attV6ConsultationCall()`, but includes an additional `consultOptions` parameter:

```
typedef enum ATTConsultOptions_t {
    CO_UNRESTRICTED = 0,
    CO_CONSULT_ONLY = 1,
    CO_TRANSFER_ONLY = 2,
    CO_CONFERENCE_ONLY = 3
} ATTConsultOptions_t;

attV10ConsultationCall(ATTPrivateData_t      *privateData,
                      DeviceID_t
                      *destRoute,
                      Boolean
                      priorityCalling,
                      ATTUserToUserInfo_t
                      userInfo,
                      ATTConsultOptions_t
                      consultOptions);
```

The supported consult options are:

- Consult Only (CO_CONSULT_ONLY) – Indicates that the intended purpose of the consultation call is to consult with the called party.
- Transfer Only (CO_TRANSFER_ONLY) – Indicates that the intended purpose of the consultation call is to set up a transfer.

- **Conference Only** (`CO_CONFERENCE_ONLY`) – Indicates that the intended purpose of the consultation call is to set up a conference.
- **Unrestricted** (`CO_UNRESTRICTED`) – Indicates that the consultation call may be used for any purpose.

The specified consult option does not actually restrict subsequent handling of the consultation call. For example, an application that has specified the consult option Transfer Only may use the consultation call to set up a conference rather than a transfer.

The consult option does affect the value of the private data Consult Mode field in the CSTA [Held Event](#), [Service Initiated Event](#), and [Originated Event](#) resulting from the consultation call:

- When the consult option is Consult Only or Unrestricted, the TSAPI Service provides private data with the subsequent Held, Service Initiated, and Originated events with a consult mode value of `ATT_CM_CONSULTATION`.

Beginning with private data version 10, the TSAPI Service also provides private data with the subsequent Held, Service Initiated, and Originated events where the consult mode value is `ATT_CM_CONSULTATION` when no consult options have been specified. (For example, when the `cstaConsultationCall()` service is invoked without private data, or when the `cstaConsultationCall()` service is invoked with private data from a pre-existing formatting function, such as `attV6ConsultationCall()`.)

(Prior to private data version 10, the consult mode value `ATT_CM_CONSULTATION` was only provided in the private data accompanying the CSTA Originated event.)

- When the consult option is Transfer Only, the TSAPI Service provides private data with the subsequent Held, Service Initiated, and Originated events with a consult mode value of `ATT_CM_TRANSFER`.
- When the consult option is Conference Only, the TSAPI Service provides private data with the subsequent Held, Service Initiated, and Originated events with a consult mode value of `ATT_CM_CONFERENCE`.

Enhanced Station Status Query

Private data version 10 supports a new escape service, `attV10QueryStationStatus()`. This escape service is similar to the pre-existing escape service `attQueryStationStatus()`, but its confirmation event also includes the station's service state (in-service or out-of-service), if available:

```
attV10QueryStationStatus(ATTPrivateData_t *privateData,
                        DeviceID_t
                        *device);

typedef enum ATTSERVICESTATE_t {
    SS_UNKNOWN = -1,
    SS_OUT_OF_SERVICE = 0,
    SS_IN_SERVICE = 1
} ATTSERVICESTATE_t;
```

```
typedef struct ATTQueryStationStatusConfEvent_t {  
    unsigned char          stationStatus;  
    ATTSERVICESTATE_t serviceState;  
} ATTQueryStationStatusConfEvent_t;
```

To receive a service state other than `SS_UNKNOWN` in the confirmation event:

- Avaya Communication Manager must be running Release 6.2 or later;
- The TSAPI CTI Link must be administered with ASAI Link Version 5 or later; and
- The station extension being queried must not correspond to a SIP endpoint.



NOTE:

Avaya Communication Manager Release 6.2 does not provide the service state of SIP endpoints. Beginning with Avaya Communication Manager Release 6.3, the service state is provided for monitored SIP endpoints.

Private Data Version 9 Features

AE Services Release 6.1 provides the following new features with Private Data Version 9:

- [Consult Mode for Held, Service Initiated, and Originated Events](#)
- [UCID in Single Step Transfer Call Confirmation Event](#)

Consult Mode for Held, Service Initiated, and Originated Events

Private data version 9 adds private data to the CSTA Held event and augments the private data for the CSTA Originated and CSTA Service Initiated events. Beginning with private data version 9:

- **The private data that may accompany any of these events will indicate if the event occurred as a result of a Conference or Transfer button press at the telephone set. (For DCP and H.323 stations, this capability requires Avaya Communication Manager Release 6.0.1 with Service Pack 1, or later. For SIP endpoints, this capability requires Avaya Communication Manager Release 6.2 or later.) If so, the consult mode reported in private data will be set to ATT_CM_CONFERENCE or ATT_CM_TRANSFER, as appropriate.**
- **The private data that may accompany the CSTA Originated event will indicate if the event occurred as a result of the `cstaConsultationCall()` service. If so, the consult mode reported in private data will be set to ATT_CM_CONSULTATION.**

```
typedef enum ATTConsultMode_t {
    ATT_CM_NONE = 0,
    ATT_CM_CONSULTATION = 1,
    ATT_CM_TRANSFER = 2,
    ATT_CM_CONFERENCE = 3,
    ATT_CM_NOT_PROVIDED = 4
} ATTConsultMode_t;

typedef struct ATTHeldEvent_t {
    ATTConsultMode_t      consultMode;
} ATTHeldEvent_t;

typedef struct ATTSserviceInitiatedEvent_t {
    ATTUCID_t             ucid;
    ATTConsultMode_t      consultMode;
} ATTSserviceInitiatedEvent_t;

typedef struct ATTOriginatedEvent_t {
    DeviceID_t             logicalAgent;
    ATTUserToUserInfo_t    userInfo;
    ATTConsultMode_t      consultMode;
} ATTOriginatedEvent_t;
```

UCID in Single Step Transfer Call Confirmation Event

A Universal Call ID (UCID) is a unique call identifier across all switches in the network. When one call is transferred to another, Avaya Communication Manager selects the UCID from one of the two calls as the UCID for the new, transferred call.

For private data version 8, an application that performed the Single Step Transfer Call service could obtain the UCID for the new call from private data in the CSTA Transferred event, but not from the Single Step Transfer Call confirmation event. This required the application to monitor the transferring device.

Beginning with private data version 9, an application that performs the Single Step Transfer Call service can obtain the UCID for the new call directly from the Single Step Transfer Call confirmation event.

```
typedef char          ATTUCID_t[64];

typedef struct ATTSingleStepTransferCallConfEvent_t {
    ConnectionID_t      transferredCall;
    ATTUCID_t           ucid;
} ATTSingleStepTransferCallConfEvent_t;
```

Private Data Version 8 Features

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

- Single Step Transfer Call – see [Single Step Transfer Call](#) on page 202.
- Calling Device in Failed Event – see [Calling Device in Failed Event](#) on page 202.
- New Get API Capabilities confirmation event – see [CSTA Get API Capabilities Private Data Versions 8 – 10 Syntax](#) on page 1032.
- A new private data parameter, flowPredictiveCallEvents, for the CSTAMonitor-CallsViaDevice service. For more information, see [Monitor Calls Via Device Service](#) on page 591.

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 419.

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 [Sample code for requesting private data](#) on page 186.

- 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 versions 5 through 15.

Sample code for requesting private data

```

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

/* Define local variables */
RetCode_t      rc;          /* Function return code */
ACSHandle_t    acsHandle;   /* ACS handle from Open Stream request */
ATTPrivateData_t privateData; /* Buffer for private data version
                               * negotiation */

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

/*
 * Now encode the requested private data versions.
 * The parameters below specify that any of the private data versions
 * in the range 5 through 15 are acceptable to this application.
 * Note that private data accompanying the ACS Open Stream
Confirmation
 * event will indicate specifically which version was negotiated for
 * this stream.
 */
if (attMakeVersionString("5-15", &privateData.data[1]) > 0)
{
    /* attMakeVersionString() succeeded */
    privateData.length = strlen(&privateData.data[1]) + 2;
}
else
{
    /* attMakeVersionString() failed */
    privateData.length = 0;
}

/* Ask to open a TSAPI Service stream with private data */
rc = acsOpenStream(
    &acsHandle,
    LIB_GEN_ID,          /* let the library generate invoke IDs */
    (Invoke_id_t)0,      /* send '0' (arbitrary) as the Invoke ID */
    ST_CSTA,             /* stream type */
    &serverID,           /* TLINK name like "AVAYA#CM1#CSTA#SERVER1"
 */
    &loginID,            /* login ID for authentication*/
    &passwd,             /* password */
    (AppName_t *) "MyApp", /* Application name */
    ACS_LEVEL1,          /* ACS level */
    (Version_t *) "TS2", /* requested TSAPI version */
    (WORD)0,             /* send queue size - use default size*/
    (WORD)5,             /* send queue extra buffers */
    (WORD)50,            /* receive queue size */
    (WORD)5,             /* receive queue extra buffers */
    (PrivateData_t *)&privateData /* formatted private data */
);

```

```
if (rc < 0)
{
    /* acsOpenStream() failed */
    return;
}

/* Wait for the ACS Open Stream Confirmation event */
```

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 saves LAN bandwidth that the private data would otherwise consume.

CSTA Get API Capabilities confirmation structures for Private Data Version 11 and later

The TSAPI Service provides information about version-dependent private services and events in the `CSTAGetAPICaps` Confirmation private data interface. For Private Data Version 11, the definition of `ATTGetAPICapsConfEvent_t` was updated to include three new fields:

Field	Description
<code>unsigned char queryEndpointRegistrationInfo;</code>	Indicates whether the endpoint registration information query (i.e., the <code>attQueryEndpointRegistrationInfo()</code> escape service) is available.
<code>unsigned char endpointRegisteredEvent;</code>	Indicates whether the Endpoint Registered private status event is available.
<code>unsigned char endpointUnregisteredEvent;</code>	Indicates whether the Endpoint Unregistered private status event is available.

Get API Capabilities Private Data Version 11 and Later Syntax

```

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

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   monitorCallsViaDevice;
    unsigned char   deviceHistoryCount;
    unsigned char   queryEndpointRegistrationInfo;
    unsigned char   endpointRegisteredEvent;
    unsigned char   endpointUnregisteredEvent;
    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 209
- [Sample Code – Set Agent State](#) on page 211
- [Sample Code – Query ACD Split](#) on page 214

Sample Code – Make Direct Agent Call

```

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

/*
 * Make Direct Agent Call - from "1000" to ACD Agent extension "1001"
 *   - ACD agent must be logged into split "2000"
 *   - 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 = "1000";   /* Call originator, an on-PBX
                                *
extension */
DeviceID_t  called  = "1001";   /* Call destination, an ACD
                                *
Agent extension */
DeviceID_t  split   = "2000";   /* ACD Agent is logged into
                                *
this split */
Boolean     priorityCall = FALSE; /* Not a priority call */
RetCode_t   rc;                /* Return code for
service

                                *
requests */
CSTAEvent_t  cstaEvent;         /* CSTA event buffer */
unsigned short eventBufSize;     /* CSTA event buffer size */
unsigned short numEvents;        /* Number of events queued
*/
ATTPrivateData_t privateData;    /* ATT service request private
                                *
data buffer */

/* Format private data for the subsequent cstaMakeCall() request */
rc = attDirectAgentCall(&privateData, &split, priorityCall, NULL);

if (rc < 0)
{
    /* Some kind of failure, handle error here. */
}

/* Invoke cstaMakeCall() with the formatted private data */

```

```

rc = cstaMakeCall(acsHandle, invokeID, &calling, &called,
                  (PrivateData_t *)&privateData);

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/* cstaMakeCall() succeeded. Wait for the confirmation event. */

/* Initialize buffer sizes before calling acsGetEventBlock() */
eventBufSize = sizeof(cstaEvent);
privateData.length = ATT_MAX_PRIVATE_DATA;

rc = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
                      &eventBufSize, (PrivateData_t *)&privateData, &numEvents);

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/* Is this the event that we are waiting for? */
if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
    (cstaEvent.eventHeader.eventType == CSTA_MAKE_CALL_CONF))
{
    if (cstaEvent.event.cstaConfirmation.invokeID == 1)
    {
        /* Invoke ID matches, cstaMakeCall() is confirmed. */
    }
    else
    {
        /* Wrong invoke ID, need to wait for another event */
    }
}
else
{
    /* Wrong event, need to wait for another event */
}

```


Sample Code – Set Agent State

```

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

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

ACSHandle_t acsHandle; /* An opened ACS
Stream Handle */
InvokeID_t invokeID = 1; /* Application-
generated invoke

    * ID */
DeviceID_t device = "1000"; /* Device associated with
ACD

    * agent */
AgentMode_t agentMode = AM_LOG_IN; /* Requested Agent Mode */
AgentID_t agentID = "3000"; /* Agent login identifier */
AgentGroup_t agentGroup = "2000"; /* ACD split to log Agent into */
AgentPassword_t *agentPassword = NULL; /* No password */
RetCode_t rc; /* Return code
for service

    * requests */
CSTAEvent_t cstaEvent; /* CSTA event buffer
*/
unsigned short eventBufSize; /* CSTA event buffer
size */
unsigned short numEvents; /* Number of events
queued */
ATTPrivateData_t privateData; /* ATT service request
private

    * data buffer */
ATTEvent_t attEvent; /* Private data event
structure */

/*
 * Format private data for the subsequent cstaSetAgentState() request
 */
rc = attV6SetAgentState(&privateData, WM_AUTO_IN, 0, TRUE);

if (rc < 0)
{
    /* Some kind of failure, handle error here. */
}

/* Invoke cstaSetAgentState() with the formatted private data */
rc = cstaSetAgentState(acsHandle, invokeID, &device, agentMode,
    &agentID, &agentGroup, agentPassword,
    (PrivateData_t *)&privateData);

```

```

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/* cstaSetAgentState() succeeded. Wait for the confirmation event. */

/* Initialize buffer sizes before calling acsGetEventBlock() */
eventBufSize = sizeof(cstaEvent);
privateData.length = ATT_MAX_PRIVATE_DATA;

rc = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
    &eventBufSize, (PrivateData_t *)&privateData, &numEvents);

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/* Is this the event that we are waiting for? */
if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
    (cstaEvent.eventHeader.eventType == CSTA_SET_AGENT_STATE_CONF))
{
    if (cstaEvent.event.cstaConfirmation.invokeID == 1)
    {
        /* Invoke ID matches, cstaSetAgentState() is confirmed. */

        /* See if the confirmation event includes private data. */
        if (privateData.length > 0)
        {
            /*
             * The confirmation event contains private data.
             * Decode it.
             */
            if (attPrivateData(&privateData, &attEvent) !=

                                ACSPOSITIVE_ACK)
            {
                /* Handle decoding error here. */
            }

            if (attEvent.eventType == ATT_SET_AGENT_STATE_CONF)
            {
                /*
                 * See whether the requested change is pending
                 */
                ATTSetAgentStateConfEvent_t
*setAgentStateConf;
                setAgentStateConf =
&privateData.u.setAgentState;
                if (setAgentStateConf->isPending == TRUE)
                {
                    /* The request is pending */
                }
            }
        }
    }
}

```

}

Sample Code – Query ACD Split

```

#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 device = "1000"; /* Device associated with
ACD

    * agent */
RetCode_t rc; /* Return code
for service

    * requests */
CSTAEvent_t cstaEvent; /* CSTA event buffer
*/
unsigned short eventBufSize; /* CSTA event buffer
size */
unsigned short numEvents; /* Number of events
queued */
ATTPrivateData_t privateData; /* ATT service request
private

    * data buffer */
ATTEvent_t attEvent; /* Private data event
structure */
ATTQueryAcdSplitConfEvent_t /* Query ACD Split
confirmation

    *queryAcdSplitConf; * event pointer
*/

/*
 * Format private data for the subsequent cstaEscapeService() request
 */
rc = attQueryAcdSplit(&privatedata, &deviceID);

if (rc < 0)
{
    /* Some kind of failure, handle error here. */
}

/* Invoke cstaEscapeService() with the formatted private data */
rc = cstaEscapeService(acsHandle, invokeID,
    (PrivateData_t *)&privateData);

if (rc != ACSPOSITIVE_ACK)
{

```

```

        /* Some kind of failure, handle error here. */
    }

    /*
     * cstaEscapeService() succeeded. Now wait for the confirmation
     event.
     *
     * To retrieve private data accompanying the confirmation event,
     * the application must provide a pointer to a private data buffer as
     * a parameter to either an acsGetEventBlock() or acsGetEventPoll()
     * request. After receiving an event, the application passes the
     * address of the private data buffer to attPrivateData() for
     decoding.
     */

    /* Initialize buffer sizes before calling acsGetEventBlock() */
    eventBufSize = sizeof(cstaEvent);
    privateData.length = ATT_MAX_PRIVATE_DATA;

    rc = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
        &eventBufSize, (PrivateData_t *)&privateData, &numEvents);

    if (rc != ACSPOSITIVE_ACK)
    {
        /* Some kind of failure, handle error here. */
    }

    /* Is this the event that we are waiting for? */
    if ((cstaEvent.eventHeader.eventClass == CSTA_CONFIRMATION) &&
        (cstaEvent.eventHeader.eventType == CSTA_ESCAPE_SVC_CONF))
    {
        if (cstaEvent.event.cstaConfirmation.invokeID == 1)
        {
            /* Invoke ID matches, cstaEscapeService() is confirmed. */

            /* See if the confirmation event includes private data. */
            if (privateData.length > 0)
            {
                /*
                 * The confirmation event contains private data.
                 * Decode it.
                 */
                if (attPrivateData(&privateData, &attEvent) !=
                    ACSPOSITIVE_ACK)
                {
                    /* Handle decoding error here. */
                }

                if (attEvent.eventType == ATT_QUERY_ACD_SPLIT_CONF)
                {
                    queryAcdSplitConf =
                        (ATTQueryAcdSplitConfEvent_t *)
                            &attEvent.u.queryAcdSplit;

                    /* Process event field values here */
                }
            }
        }
    }

```

```
        }  
    }  
    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.

- [Using the private data header files](#) on page 219
- [The attpdefs.h file – PDU names and numbers](#) on page 219
- [The attpriv.h file – other related PDU elements](#) on page 220
- [Upgrading PDV 10 applications to PDV 1](#) on page 227
- [Upgrading PDV 9 applications to PDV 1](#) on page 229
- [Upgrading PDV 8 applications to PDV 1](#) on page 231
- [Upgrading PDV 7 applications to PDV 1](#) on page 233
- [Upgrading PDV 6 applications to PDV 1](#) on page 234
- [Maintaining a PDV 12 application in a PDV 13 environment](#)

To maintain a PDV 12 application in an AE Services 7.1.1 PDV 13 environment, you will need to make coding level changes to your application, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for maintaining a PDV 12 application in a PDV 13 environment.

1. Make sure you have installed the AE Services 7.1.1 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_DELIVERED ATTDeliveredEvent_t deliveredEvent ATT_ESTABLISHED ATTEstablishedEvent_t establishedEvent	ATTV12_DELIVERED ATTV12DeliveredEvent_t v12deliveredEvent ATTV12_ESTABLISHED ATTV12EstablishedEvent_t v12establishedEvent

3. Recompile your application with AE Services 7.1.1 TSAPI SDK.

Things you do not need to change in your code

If you are maintaining a PDV 12 application in a PDV 13 environment, you do not need to change the open stream request. Your application will continue to negotiate a PDV 12 stream.

- Maintaining a PDV 11 application in a PDV 13 environment on page 235
- [Maintaining a PDV 10 application in a PDV 13 environment](#) on page 237
- [Maintaining a PDV 9 application in a PDV 13 environment](#) on page 239
- [Maintaining a PDV 8 application in a PDV 13 environment](#) on page 240
- [Maintaining a PDV 7 application in a PDV 13 environment](#) on page 241
- [Recompiling against the same SDK](#) on page 242



NOTE:

The AE Services 6.3.3 TSAPI Service is at PDV 12. Any new TSAPI applications developed with the AE Services Release 6.3.3 TSAPI SDK should be written to include support for PDV 12.

Using the private data header files

The private data header files (`attpdefs.h` and `attppriv.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_FAILED 141` – the private PDU value representing the highest private data version for the Failed event, which is Private Data Versions 8 through 10. (This event was changed for PDV 8, but did not change for PDV 9 or PDV 10).

`#define ATTV7_FAILED 137` – the private PDU value representing the previous private data version for the Failed event, which is Private Data Version 7. (There was no private data associated with the Failed event prior to PDV 7.)

`#define ATT_QUERY_DEVICE_NAME_CONF 125` – the private PDU value representing the highest private data version of the Query Device Name confirmation event, which is Private Data Version 7. (This event did not change for Private Data Versions 8 through 10.)

`#define ATTV6_QUERY_DEVICE_NAME_CONF 89` – the private PDU value representing the previous private data versions of the Query Device Name service, which is Private Data Versions 5 and 6.

`#define ATTV4_QUERY_DEVICE_NAME_CONF 50` – the private PDU value representing the previous private data versions of the Query Device Name service, which is Private Data Versions 2 through 4.

`#define ATT_ROUTE_SELECT 126` – the private PDU value representing highest private data version of the Route Select service, which is Private Data Versions 7 through 10. (This service was changed for PDV 7, but did not change for Private Data Versions 8 through 10.)

`#define ATTV6_ROUTE_SELECT 116` – the private PDU value representing the previous private data version of the Route Select service, PDV 6.

`#define ATTV5_ROUTE_SELECT 43` – the private PDU value representing the previous private data versions of the Route Select service, which is Private Data Versions 2 through 5.

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 an earlier version (or versions) 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 an earlier version (or versions) 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 an earlier version (or versions) 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
<code>ATT_QUERY_DEVICE_NAME_CONF 125</code>	• <code>ATTQueryDeviceNameConfEvent_t</code> (structure name) • <code>queryDeviceName</code> (union member name)
<code>ATTV6_QUERY_DEVICE_NAME_CONF 89</code>	• <code>ATTV6QueryDeviceNameConfEvent_t</code> (structure name) • <code>v6queryDeviceName</code> (union member name)
<code>ATT_ROUTE_SELECT 126</code>	• <code>ATTRouteSelect_t</code> (structure name) • <code>routeSelectReq</code> (union member name) • <code>attV7RouteSelect()</code> (function name)

Table 12: Elements in private data header files

PDU name and number attpdefs.h	Related elements in attpriv.h
ATTV6_ROUTE_SELECT 116	• ATTV6RouteSelect_t (structure name) • v6routeSelectReq (union member name) • attV6RouteSelect() (function name)

Upgrading PDV 13 applications to PDV 15

If you have an existing application that was developed to PDV 13 and you want to take advantage of PDV 15 functionality, you will need to update your application to take advantage of the PDV 15 features, and then recompile your application with the AE Services 8.1 TSAPI SDK. The following steps outline the high level tasks necessary for upgrading a PDV 13 application to PDV 15:

1. Make sure you have installed the AE Services 8.1 TSAPI SDK. Whenever you compile or recompile your PDV 15 applications, you must use the AE Services 8.1 (or later) TSAPI SDK, which supports PDV 15.
2. Use on page 1027 to help you determine what PDV 15 functionality you want to incorporate into your application.
3. Make the following coding level changes to your application:
 - ATT Originated Event private data:
 - Replace all references to event type `ATT_ORIGINATED` with event type `ATTV14_ORIGINATED`.
 - Replace all references to structure name `ATTOriginatedEvent_t` with structure name `ATTV14OriginatedEvent_t`.
 - Replace all references to the `ATTEvent_t` union member `u.OriginatedEvent` with `u.v14OriginatedEvent`.
 - Add logic to handle private data with event type `ATT_ORIGINATED` that may accompany a CSTA Originated event.
 - ATT MakeCall private data:
 - Replace all references to event type `ATT_MAKE_CALL` with event type `ATTV14_MAKE_CALL`.
 - Replace all references to structure name `ATTMakeCall_t` with structure name `ATTV14MakeCall_t`.
 - Replace all references to the `ATTEvent_t` union member `u.makeCallReq` with `u.v14makeCallReq`.
 - Add logic to handle private data with event type `ATT_MAKE_CALL` that may accompany a CSTA MakeCall.
 - ATT AnswerCall private data:
 - Replace all references to event type `ATT_ANSWER_CALL` with event type `ATTV14_ANSWER_CALL`.
 - Replace all references to structure name `ATTAnswerCall_t` with structure name `ATTV14AnswerCall_t`.
 - Replace all references to the `ATTEvent_t` union member `u.answerCallReq` with `u.v14answerCallReq`.
 - Add logic to handle private data with event type `ATT_ANSWER_CALL` that may accompany a CSTA AnswerCall.

- ATT ConsultationCall private data:
 - Replace all references to event type `ATT_CONSULTATION_CALL` with event type `ATTV14_CONSULTATION_CALL`.
 - Replace all references to structure name `ATTConsultationCall_t` with structure name `ATTV14ConsultationCall_t`.
 - Replace all references to the `ATTEvent_t` union member `u.consultationCallReq` with `u.v14consultaitonCallReq`.
 - Add logic to handle private data with event type `ATT_CONSULTAION_CALL` that may accompany a CSTA ConsultationCall.

 - ATT SingleStepTransferCall private data:
 - Replace all references to event type `ATT_SINGLE_STEP_TRANSFER_CALL` with event type `ATTV14_SINGLE_STEP_TRANSFER_CALL`.
 - Replace all references to structure name `ATTSingleStepTransferCall_t` with structure name `ATTV14SingleStepTransferCall_t`.
 - Replace all references to the `ATTEvent_t` union member `u.singleStepTransferCallReq` with `u.v14singleStepTransferCallReq`.
 - Add logic to handle private data with event type `ATT_SINGLE_STEP_TRANSFER_CALL` that may accompany a CSTA SingleStepTransferCall.
4. Change `acsOpenStream` to negotiate PDV 15. See [Requesting private data](#) on page 185.
 5. Recompile your application with the AE Services 8.1 TSAPI SDK, which supports private data versions 2 through 15.

Upgrading PDV 12 applications to PDV 13

If you have an existing application that was developed to PDV 11 and you want to take advantage of PDV 13 functionality, you will need to update your application to take advantage of the PDV 13 features, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for upgrading a PDV 12 application to PDV 13:

1. Make sure you have installed the AE Services 7.1.1 TSAPI SDK. Whenever you compile or recompile your PDV 13 applications, you must use the AE Services 7.1.1 (or later) TSAPI SDK, which supports PDV 13.

Private Data Version 12 features Private Data Version 13 features

The TSAPI Service supports private data version 13 beginning with AE Services Release 7.1.1. Private data version 13 provides the following new feature:

Channel Type in Delivered Event

Beginning with private data version 13, the private data accompanying the CSTA [Diverted Event](#) includes the Channel Type, if available.

Channel Type in Established Event

Beginning with private data version 13, the private data accompanying the CSTA [Diverted Event](#) includes the Channel Type, if available.

2. Private Data Version 12 features on page 1027 to help you determine what PDV 13 functionality you want to incorporate into your application.
3. Make the following coding level changes to your application:
 - ATT Delivered Event private data:
 - Replace all references to event type `ATT_DELIVERED` with event type `ATTV12_DELIVERED`.
 - Replace all references to structure name `ATTDeliveredEvent_t` with structure name `ATTV12DeliveredEvent_t`.
 - Replace all references to the `ATTEvent_t` union member `u.deliveredEvent` with `u.v12deliveredEvent`.)
 - Add logic to handle private data with event type `ATT_DELIVERED` that may accompany a CSTA Delivered event.
 - ATT Established Event private data:
 - Replace all references to event type `ATT_ESTABLISHED` with event type `ATTV12_ESTABLISHED`.
 - Replace all references to structure name `ATTEstablishedEvent_t` with structure name `ATTV12EstablishedEvent_t`.
 - Replace all references to the `ATTEvent_t` union member `u.establishedEvent` with `u.v12establishedEvent`.)
 - Add logic to handle private data with event type `ATT_ESTABLISHED` that may accompany a CSTA Delivered event.

4. Change `acsOpenStream` to negotiate PDV 13. See [Requesting private data](#) on page 185.
5. Recompile your application with the AE Services 7.1.1 TSAPI SDK, which supports private data versions 2 through 13.

Upgrading PDV 11 applications to PDV 13

If you have an existing application that was developed to PDV 11 and you want to take advantage of PDV 13 functionality, you will need to update your application to take advantage of the PDV 13 features, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for upgrading a PDV 11 application to PDV 13:

1. Make sure you have installed the AE Services 7.1.1 TSAPI SDK. Whenever you compile or recompile your PDV 13 applications, you must use the AE Services 7.1.1 (or later) TSAPI SDK, which supports PDV 13.
2. Use [Private Data Version 13 features](#)

The TSAPI Service supports private data version 13 beginning with AE Services Release 7.1.1. Private data version 13 provides the following new feature:

Channel Type in Delivered Event

Beginning with private data version 13, the private data accompanying the CSTA [Diverted Event](#) includes the Channel Type, if available.

Channel Type in Established Event

Beginning with private data version 13, the private data accompanying the CSTA [Diverted Event](#) includes the Channel Type, if available.

3. Private Data Version 12 features on page 1027 to help you determine what PDV 12 functionality you want to incorporate into your application.
4. Make the following coding level changes to your application:
 - ATT Diverted Event private data:
 - Replace all references to event type `ATT_DIVERTED` with event type `ATTV11_DIVERTED`.
 - Replace all references to structure name `ATTDivertedEvent_t` with structure name `ATTV11DivertedEvent_t`.
 - Replace all references to the `ATTEvent_t` union member `u.diverted-Event` with `u.v11divertedEvent`.)
 - Add logic to handle private data with event type `ATT_DIVERTED` that may accompany a CSTA Diverted event.
5. Change `acsOpenStream` to negotiate PDV 13. See [Requesting private data](#) on page 185.
6. Recompile your application with the AE Services 7.1.1 TSAPI SDK, which supports private data versions 2 through 13.

Upgrading PDV 10 applications to PDV 13

If you have an existing application that was developed to PDV 10 and you want to take advantage of newer private data features, you will need to update your application to take advantage of those features, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for upgrading a PDV 10 application to PDV 13:

1. Make sure you have installed the AE Services 7.1.1 TSAPI SDK. Whenever you compile or recompile your PDV 13 applications, you must use the AE Services 7.1.1 (or later) TSAPI SDK, which supports PDV 13.
2. Use [Private Data Version 11 features](#) on page 1000 to help you determine what PDV 11 functionality you want to incorporate into your application.
3. Make the following coding level changes to your application:
 - ATT Get API Caps private data:
 - Replace all references to event type `ATT_GETAPI_CAPS_CONF` with event type `ATTV10_GETAPI_CAPS_CONF`. (The unversioned event type `ATT_GETAPI_CAPS_CONF` corresponds to the highest private data version. For AE Services 6.3.1, the highest private data version is 11.)
 - Replace all references to structure name `ATTGetAPICapsConfEvent_t` with structure name `ATTV10GetAPICapsConfEvent_t`. (The unversioned structure name `ATTGetAPICapsConfEvent_t` corresponds to the highest private data version, now 11.)
 - Replace all references to the `ATTEvent_t` union member `u.getAPICaps` with `u.v10getAPICaps`. (The unversioned union member `u.getAPICaps` corresponds to the highest private data version, now 11.)
 - Add logic to handle private data with event type `ATT_GETAPI_CAPS_CONF` that may accompany a CSTA Get API Caps confirmation event.
 - ATT Query Registration Info service:
 - Add logic to invoke the `attQueryEndpointRegistrationInfo()` escape service, as desired, and to handle private data with event type `ATT_QUERY_ENDPOINT_REGISTRATION_INFO_CONF` that will accompany the corresponding CSTA Escape Service confirmation event.
 - ATT Endpoint Registered events:
 - Add logic to handle a CSTA Private Status event with private data event type `ATT_ENDPOINT_REGISTERED`. Or, if your application does not wish to receive Endpoint Registered events, then use the `attMonitorFilterExt()` private data formatting function to set the `ATT_ENDPOINT_REGISTERED_FILTER` flag before invoking `cstaMonitorDevice()` for station extensions.
 - ATT Endpoint Unregistered events:
 - Add logic to handle a CSTA Private Status event with private data event type `ATT_ENDPOINT_UNREGISTERED`. Or, if your application does not wish to receive Endpoint Unregistered events, then use the `attMonitorFilterExt()` private data formatting function to set the `ATT_ENDPOINT_UNREGISTERED_FILTER` flag before invoking `cstaMonitorDevice()` for station extensions.

`UNREGISTERED_FILTER` flag before invoking `cstaMonitorDevice()` for station extensions.

- CSTA Failed events:
 - Ensure that your application is able to handle a CSTA Failed event where:
 - the `deviceId` in the `failedConnection` is the empty string (“”)
 - the `deviceId` of the `failingDevice` is the empty string (“”) and has device ID status `ID_NOT_KNOWN`
 - the `olddeviceId` in the device history is “Not Known”
 - the `deviceId` of the `oldconnectionID` in the device history is “Not Known”.

Upgrading PDV 9 applications to PDV 13

If you have an existing application that was developed to PDV 9 and you want to take advantage of newer private data features, you will need to update your application to take advantage of those features, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for upgrading a PDV 9 application to PDV 13:

1. Make sure you have installed the AE Services 7.1.1 TSAPI SDK. Whenever you compile or recompile your PDV 13 applications, you must use the AE Services 7.1.1 TSAPI SDK, which supports PDV 13.
2. Use [Private Data Version 10 features](#) on page 1002 to help you determine what PDV 10 functionality you want to incorporate into your application.
3. Make the following coding level changes to your application:
 - ATT Consultation Call private data:
 - Replace all references to event type `ATT_CONSULTATION_CALL` with event type `ATTV9_CONSULTATION_CALL`.
 - Replace all references to structure name `ATTConsultationCall_t` with structure name `ATTV9ConsultationCall_t`.
 - Replace all references to the `ATTEvent_t` union member `u.consultationCallReq` with `u.v9consultationCallReq`.
 - Add logic to use the private data formatting function `attV10ConsultationCall()` as desired.
 - ATT Query Station Status private data:
 - Replace all references to event types `ATT_QUERY_STATION_STATUS` and `ATT_QUERY_STATION_STATUS_CONF` with event types `ATTV9_QUERY_STATION_STATUS` and `ATTV9_QUERY_STATION_STATUS_CONF`, respectively.
 - Add logic to handle private data with event type `ATT_QUERY_STATION_STATUS_CONF` that may accompany a CSTA Escape Service confirmation event.

NOTE:

Recall that the confirmation event for a request is always in the format of the negotiated private data version, even if the request is sent in the format of a previous private data version. So, for example, if your application negotiates private data version 10 and invokes the `attQueryStationStatus()` escape service rather than the `attV10QueryStationStatus()` escape service, the application will receive a CSTA Escape Service Confirmation event with private data event type `ATT_QUERY_STATION_STATUS_CONF`; not `ATTV9_QUERY_STATION_STATUS_CONF`. Your application should maintain logic for event type `ATTV9_QUERY_STATION_STATUS_CONF` in case it is only able to negotiate a private data version less than 10.

- Replace all references to structure names `ATTQueryStationStatus_t` and `ATTQueryStationStatusConfEvent_t` with structure names `ATTV9-`

`QueryStationStatus_t` and `ATTV9QueryStationStatusConfEvent_t`, respectively.

- Replace all references to the `ATTEvent_t` union members `u.queryStationStatusReq` and `u.queryStationStatus` with `u.v9queryStationStatusReq` and `u.v9queryStationStatus`, respectively.
 - Add logic to use the new private data formatting function `attV10QueryStationStatus()` as desired.
4. Continue with the procedure [Upgrading PDV 10 applications to PDV 1](#) on page 207.

Upgrading PDV 8 applications to PDV 13

If you have an existing application that was developed to PDV 8, and you want to take advantage of newer private data features, you will need to update your application to take advantage of those features, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for upgrading a PDV 8 application to PDV 13:

1. Make sure you have installed the AE Services 7.1.1 TSAPI SDK. Whenever you recompile your PDV 13 applications, you must use the AE Services 7.1.1 TSAPI SDK, which supports PDV 13.
2. Use [Private Data Version 9 features](#) on page 1005 to help you determine what PDV 9 functionality you want to incorporate into your application.
3. Make the following coding level changes to your application:
 - ATT Held Event private data:
 - Add logic to handle private data with event type `ATT_HELD` that may accompany a CSTA Held event.
 - ATT Originated Event private data:
 - Replace all references to event type `ATT_ORIGINATED` with event type `ATTV8_ORIGINATED`.
 - Replace all references to structure name `ATTOriginatedEvent_t` with structure name `ATTV8OriginatedEvent_t`.
 - Replace all references to the `ATTEvent_t` union member `u.OriginatedEvent` with `u.v8originatedEvent`.)
 - Add logic to handle private data with event type `ATT_ORIGINATED` that may accompany a CSTA Originated event.
 - ATT Service Initiated Event private data:
 - Replace all references to event type `ATT_SERVICE_INITIATED` with event type `ATTV8_SERVICE_INITIATED`.
 - Replace all references to structure name `ATTServiceInitiatedEvent_t` with structure name `ATTV8ServiceInitiatedEvent_t`.
 - Replace all references to the `ATTEvent_t` union member `u.serviceInitiated` with `u.v8serviceInitiated`.
 - Add logic to handle private data with event type `ATT_SERVICE_INITIATED` that may accompany a CSTA Service Initiated event.
 - ATT Single Step Transfer Call Confirmation Event private data:
 - Replace all references to event type `ATT_SINGLE_STEP_TRANSFER_CALL_CONF` with event type `ATTV8_SINGLE_STEP_TRANSFER_CALL_CONF`.
 - Replace all references to structure name `ATTSingleStepTransferCallConfEvent_t` with structure name `ATTV8SingleStepTransferCallConfEvent_t`.
 - Replace all references to the `ATTEvent_t` union member `u.ssTransferCallConf` with `u.v8ssTransferCallConf`.

- Add logic to handle private data with event type `ATT_SINGLE_STEP_TRANSFER_CALL_CONF` that may accompany a CSTA Escape Service confirmation event.
4. Continue with the procedure [Upgrading PDV 9 applications to PDV 13](#) on page 209.

Upgrading PDV 7 applications to PDV 13

If you have an existing application that was developed to PDV 7, and you want to take advantage of newer private data features, you will need to update your application to take advantage of those features, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for upgrading a PDV 7 application to PDV 13:

1. Make sure you have installed the AE Services 7.1.1 TSAPI SDK. Whenever you recompile your PDV 13 applications, you must use the AE Services 7.1.1 TSAPI SDK, which supports PDV 13.
2. Use [Private Data Version 8 features](#) on page 1007 to help you determine what PDV 8 functionality you want to incorporate into your application.
3. Make the following coding level changes to your application:
 - ATT Failed Event private data:
 - Replace all references to event type `ATT_FAILED` with event type `ATTV7_FAILED`.
 - Replace all references to structure name `ATTFailedEvent_t` with structure name `ATTV7FailedEvent_t`.
 - Replace all references to the `ATTEvent_t` union member `u.failedEvent` with `u.v7failedEvent`.
 - Add logic to handle private data with event type `ATT_FAILED` that may accompany a CSTA Failed event.
 - Single Step Transfer Call Service:
 - Add logic to use this new escape service.
4. Continue with the procedure [Upgrading PDV 8 applications to PDV 1](#) on page 211.

Upgrading PDV 6 applications to PDV 13

If you have an existing application that was developed to PDV 6, and you want to take advantage of newer private data features, you will need to need to update your application to take advantage of those features, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for upgrading a PDV 6 application to PDV 13.

1. Make sure you have installed the AE Services 7.1.1 TSAPI SDK. Whenever you recompile your PDV 13 applications, you must use the AE Services 7.1.1 TSAPI SDK, which supports PDV 13.
2. Use [Private Data Version 7 features](#) on page 1008 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. Continue with the procedure [Upgrading PDV 7 applications to PDV 13](#) on page 213.

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 12 application in a PDV 13 environment

To maintain a PDV 12 application in an AE Services 7.1.1 PDV 13 environment, you will need to make coding level changes to your application, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for maintaining a PDV 12 application in a PDV 13 environment.

5. Make sure you have installed the AE Services 7.1.1 TSAPI SDK.
6. 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_DELIVERED ATTDeliveredEvent_t deliveredEvent ATT_ESTABLISHED ATTEstablishedEvent_t establishedEvent	ATTV12_ DELIVERED ATTV12DeliveredEvent_t v12deliveredEvent ATTV12_ ESTABLISHED ATTV12EstablishedEvent_t v12establishedEvent

7. Recompile your application with AE Services 7.1.1 TSAPI SDK.

Things you do not need to change in your code

If you are maintaining a PDV 12 application in a PDV 13 environment, you do not need to change the open stream request. Your application will continue to negotiate a PDV 12 stream.

Maintaining a PDV 11 application in a PDV 13 environment

To maintain a PDV 11 application in an AE Services 7.1.1 PDV 13 environment, you will need to make coding level changes to your application, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for maintaining a PDV 11 application in a PDV 13 environment.

8. Make sure you have installed the AE Services 7.1.1 TSAPI SDK.
9. 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_DIVERTED ATTDivertedEvent_t divertedEvent	ATTV11_DIVERTED ATTV11DivertedEvent_t v11divertedEvent

10. Recompile your application with AE Services 7.1.1 TSAPI SDK.

Things you do not need to change in your code

If you are maintaining a PDV 11 application in a PDV 13 environment, you do not need to change the open stream request. Your application will continue to negotiate a PDV 11 stream.

Maintaining a PDV 10 application in a PDV 13 environment

To maintain a PDV 10 application in an AE Services 7.1.1 PDV 13 environment, you will need to make coding level changes to your application, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for maintaining a PDV 11 application in a PDV 13 environment.

1. Make sure you have installed the AE Services 7.1.1 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_GETAPI_CAPS_CONF ATTGetAPICapsConfEvent_t getAPICaps	ATTV10_GETAPI_CAPS_CONF ATTV10GetAPICapsConfEvent_t v10getAPICaps

3. Continue with the procedure [Maintaining a PDV 12 application in a PDV 13 environment](#)

To maintain a PDV 12 application in an AE Services 7.1.1 PDV 13 environment, you will need to make coding level changes to your application, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for maintaining a PDV 12 application in a PDV 13 environment.

4. Make sure you have installed the AE Services 7.1.1 TSAPI SDK.
5. 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_DELIVERED ATTDeliveredEvent_t deliveredEvent ATT_ESTABLISHED ATTEstablishedEvent_t establishedEvent	ATTV12_ DELIVERED ATTV12DeliveredEvent_t v12deliveredEvent ATTV12_ ESTABLISHED ATTV12EstablishedEvent_t v12establishedEvent

6. Recompile your application with AE Services 7.1.1 TSAPI SDK.

Things you do not need to change in your code

If you are maintaining a PDV 12 application in a PDV 13 environment, you do not need to change the open stream request. Your application will continue to negotiate a PDV 12 stream.

7. Maintaining a PDV 11 application in a PDV 13 environment on page 227.

Things you do not need to change in your code

If you are maintaining a PDV 10 application in a PDV 13 environment, you do not need to change the open stream request. Your application will continue to negotiate a PDV 10 stream.

Maintaining a PDV 9 application in a PDV 13 environment

To maintain a PDV 9 application in an AE Services 7.1.1 PDV 13 environment, you will need to make coding level changes to your application, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for maintaining a PDV 9 application in a PDV 13 environment.

1. Make sure you have installed the AE Services 7.1.1 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_CONSULTATION_CALL ATTConsultationCall_t consultationCallReq	ATTV9_CONSULTATION_CALL ATTV9ConsultationCall_t v9consultationCallReq
ATT_QUERY_STATION_STATUS ATTQueryStationStatus_t queryStationStatusReq	ATTV9_QUERY_STATION_STATUS ATTV9QueryStationStatus_t v9queryStationStatusReq
ATT_QUERY_STATION_STATUS_CONF ATTQueryStationStatusConfEvent_t queryStationStatus	ATTV9_QUERY_STATION_STATUS_CONF ATTV9QueryStationStatusConfEvent_t v9queryStationStatus

3. Continue with the procedure [Maintaining a PDV 10 application in a PDV 13 environment](#) on page 227.

Things you do not need to change in your code

If you are maintaining a PDV 9 application in a PDV 13 environment, you do not need to change the open stream request. Your application will continue to negotiate a PDV 9 stream.

Maintaining a PDV 8 application in a PDV 13 environment

To maintain a PDV 8 application in an AE Services 7.1.1 PDV 13 environment, you will need to make coding level changes to your application, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for maintaining a PDV 8 application in a PDV 13 environment.

1. Make sure you have installed the AE Services 7.1.1 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_ORIGINATED ATTOriginatedEvent_t originatedEvent	ATTV8_ORIGINATED ATTV8OriginatedEvent_t v8originatedEvent
ATT_SERVICE_INITIATED ATTServiceInitiatedEvent_t serviceInitiated	ATTV8_SERVICE_INITIATED ATTV8ServiceInitiatedEvent_t v8serviceInitiated
ATT_SINGLE_STEP_TRANSFER_CALL_CONF ATTSingleStepTransferCallConfEvent_t ssTransferCallConf	ATTV8_SINGLE_STEP_TRANSFER_CALL_CONF ATTV8SingleStepTransferCallConfEvent_t v8ssTransferCallConf

3. Continue with the procedure [Maintaining a PDV 9 application in a PDV 13 environment](#) on page 27.

Things you do not need to change in your code

If you are maintaining a PDV 8 application in a PDV 13 environment, you do not need to change the open stream request. Your application will continue to negotiate a PDV 8 stream.

Maintaining a PDV 7 application in a PDV 13 environment

To maintain a PDV 7 application in an AE Services 7.1.1 PDV 13 environment you will need to make coding level changes to your application, and then recompile your application with the AE Services 7.1.1 TSAPI SDK. The following steps outline the high level tasks necessary for maintaining a PDV 7 application in a PDV 13 environment.

1. Make sure you have installed the AE Services 7.1.1 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. Continue with the procedure [Maintaining a PDV 8 application in a PDV 13 environment](#) on page 229.

Things you do not need to change in your code

If you are maintaining a PDV 7 application in a PDV 13 environment, you do not need to change the open stream request. Your application 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 9 SDK, you would recompile with the AE Services 6.1 (PDV 9) SDK, as opposed to the AE Services 7.1.1 (PDV 9) 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

The Call Control Service Group provides services that enable a TSAPI 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](#) on page 245
- [Alternate Call Service](#) on page 255
- [Answer Call Service](#) on page 259
- [Private Data Version](#) 15 and Later Syntax

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

/*
 * attAnswerCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attAnswerCall(
    ATTPriData_t      *privateData,
    ATTVideoCall_t    videocall);

/* specified */

typedef struct ATTPriData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPriData_t;

typedef enum ATTVideoCall_t{
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2
} ATTVideoCall_t;
```

- [Clear Call Service](#) on page 264
- [Clear Connection Service](#) on page 267
- [Conference Call Service](#) on page 276
- [Consultation Call Service](#) on page 283
- [Consultation Direct-Agent Call Service](#) on page 297
- [Consultation Supervisor-Assist Call Service](#) on page 307
- [Deflect Call Service](#) on page 317
- [Hold Call Service](#) on page 322
- [Make Call Service](#) on page 326
- [Make Direct-Agent Call Service](#) on page 342
- [Make Predictive Call Service](#) on page 352
- [Make Supervisor-Assist Call Service](#) on page 364
- [Pickup Call Service](#) on page 373
- [Reconnect Call Service](#) on page 378
- [Retrieve Call Service](#) on page 387
- [Send DTMF Tone Service \(Private Data Version 4 and Later\)](#) on page 391
- [Selective Listening Hold Service \(Private Data Version 5 and Later\)](#) on page 399
- [Selective Listening Retrieve Service \(Private Data Version 5 and Later\)](#) on page 405
- [Single Step Conference Call Service \(Private Data Version 5 and Later\)](#) on page 410
- [Single Step Transfer Call \(Private Data Version 8 and later\)](#) on page 419
- [Transfer Call Service](#) on page 427

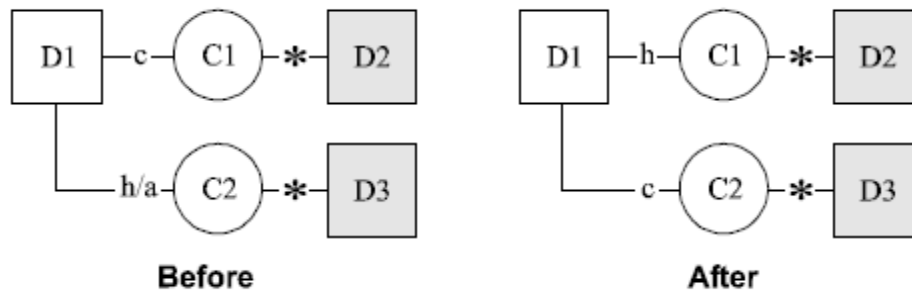
Graphical Notation Used in the Diagrams

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.

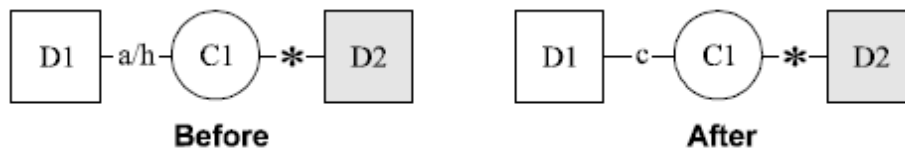
Alternate Call Service

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.



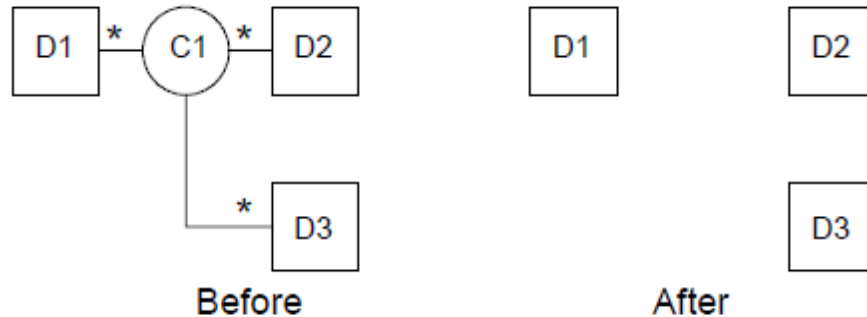
Answer Call Service

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).



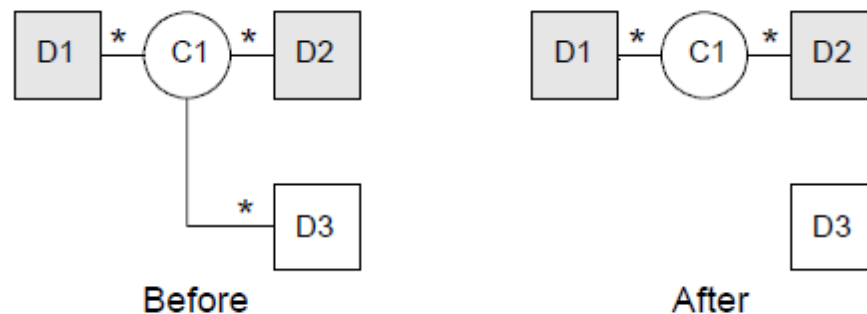
Clear Call Service

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



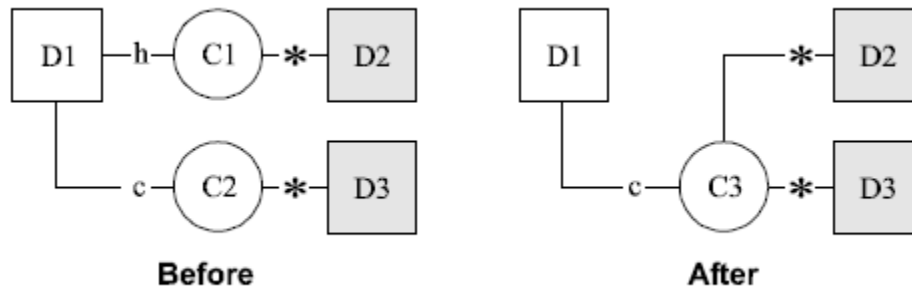
Clear Connection Service

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 switch hook, 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.



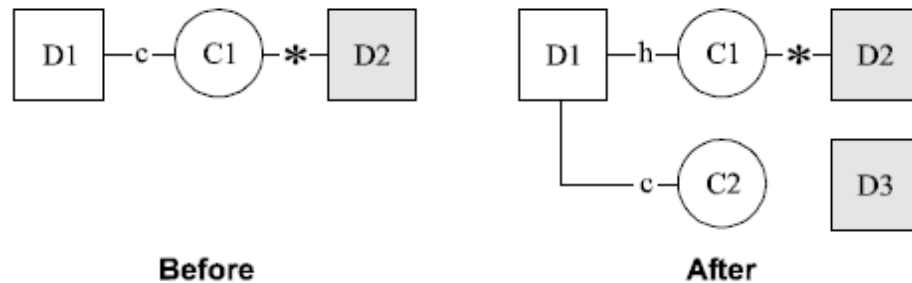
Conference Call Service

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.



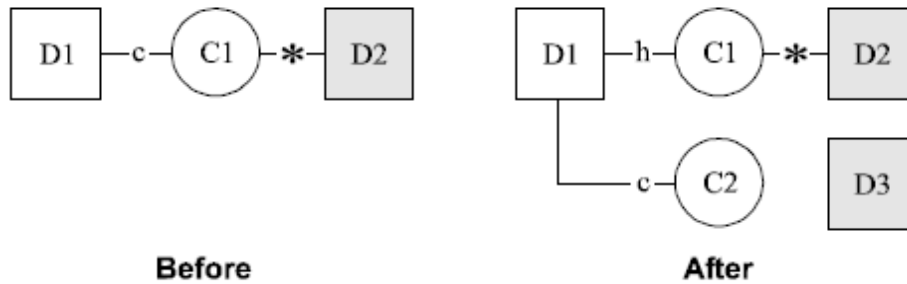
Consultation Call Service

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 result is the connection `newCall` (C2-D1).



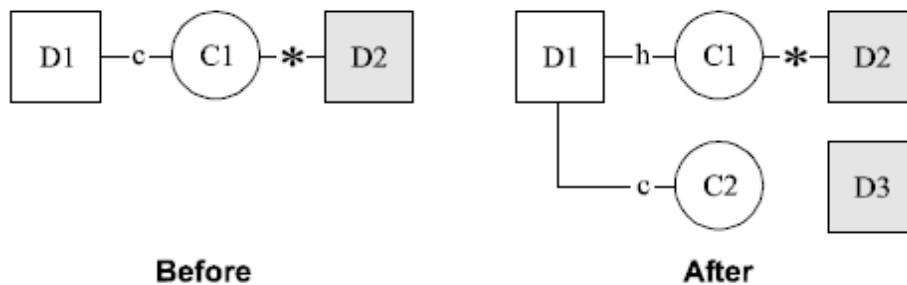
Consultation Direct-Agent Call Service

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 result is the connection `newCall` (C2-D1).



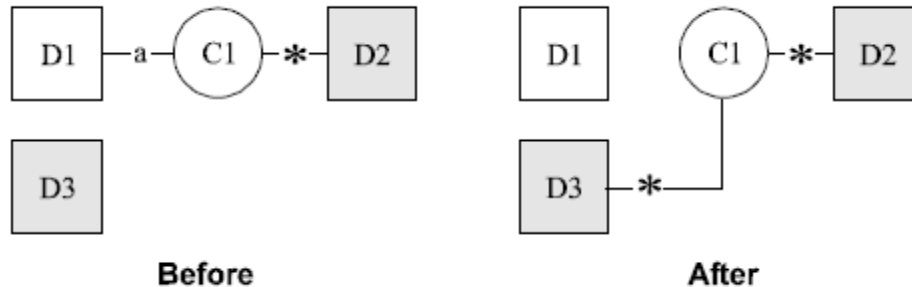
Consultation Supervisor-Assist Call Service

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 result is the connection `newCall` (C2-D1).



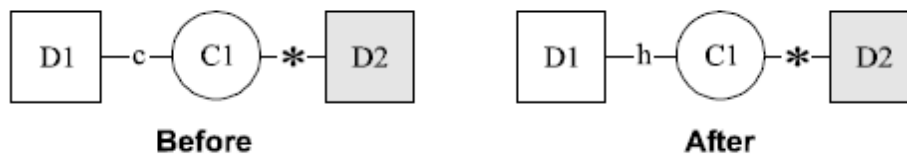
Deflect Call Service

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.



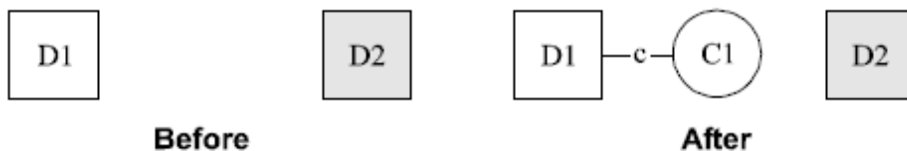
Hold Call Service

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 switch hook 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.



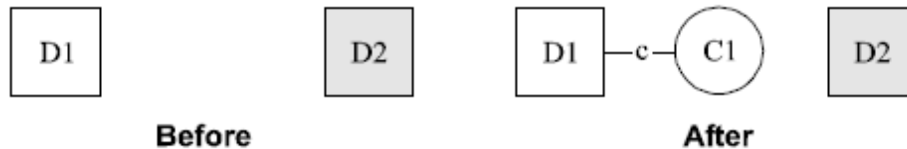
Make Call Service

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 result is 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 result is the connection `newCall` (C1-D1).



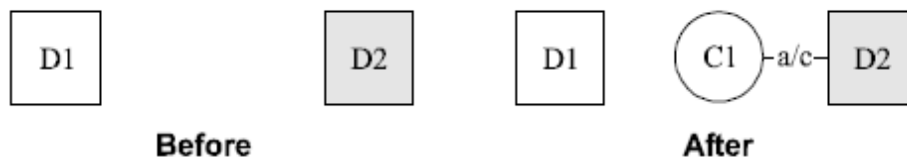
The Make Direct Agent Call Service should be used only in the following two situations:

- Direct Agent Calls in a non-EAS environment
- Direct Agent Calls in an Expert Agent Selection (EAS) environment only when it is required to ensure that these calls against a skill other than that skill specified for these measurements on Avaya Communication Manager 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 Avaya Communication Manager for that agent.

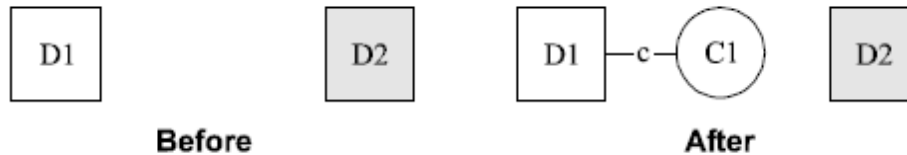
Make Predictive Call Service

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).



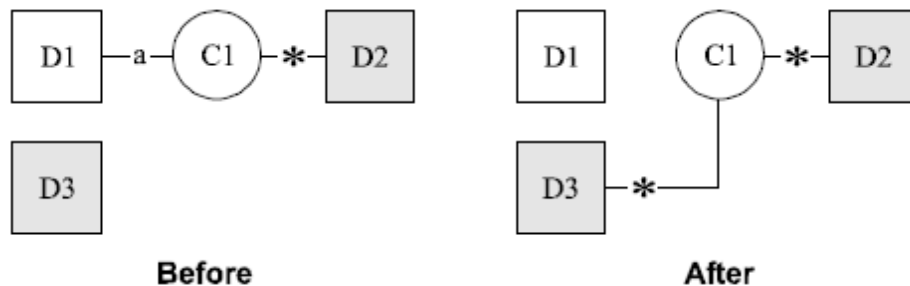
Make Supervisor-Assist Call Service

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 result is the connection `newCall` (C1-D1).



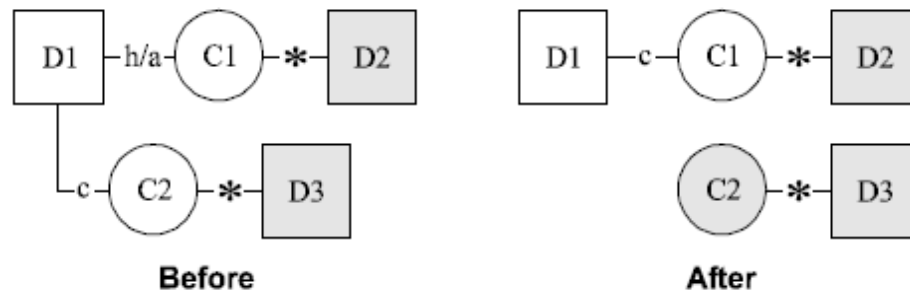
Pickup Call Service

The Pickup Call Service takes an alerting call (C1) at a device (D1) with the connection `deflectCall` to another on-PBX device.



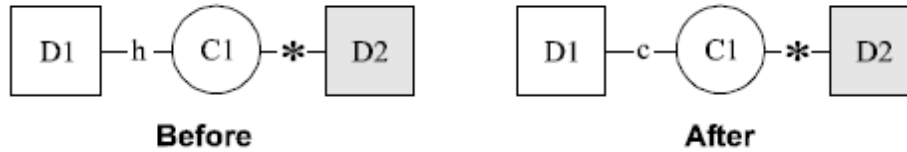
Reconnect Call Service

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).



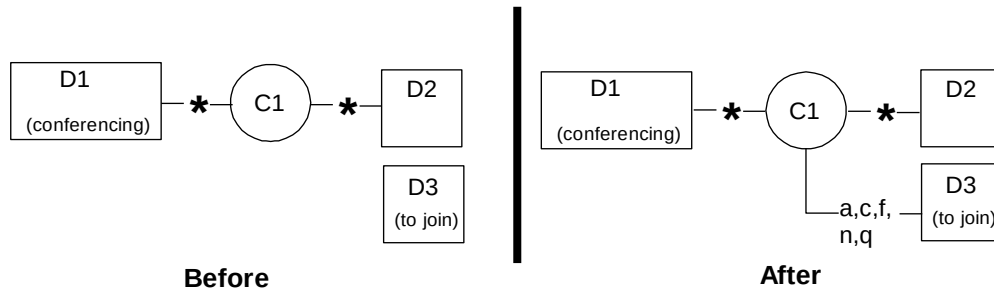
Retrieve Call Service

The service restores a held connection `heldCall` (C1-D1) to the Connected state (active).



Single Step Conference Call

The Single Step Conference Call service collapses the two steps of the conference call process into one.

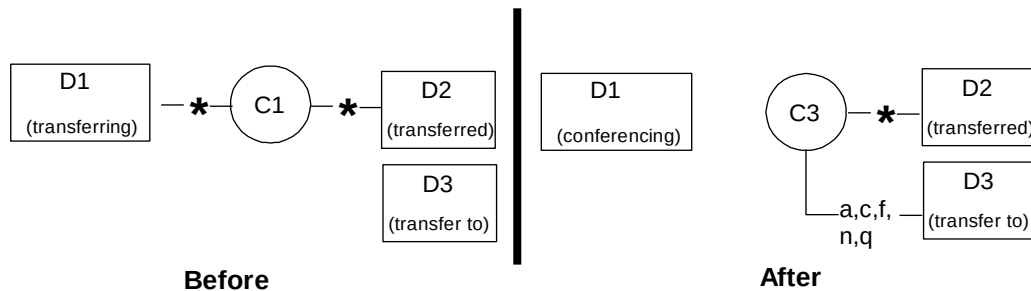


By specifying D3 as the destination for a single step conference involving call C1, the connection D3C1 is created in exactly the same way as if any of the devices already in C1 had placed a new call to D3 using the Make Call service. The difference is that all of the devices already in C1 remain in 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.

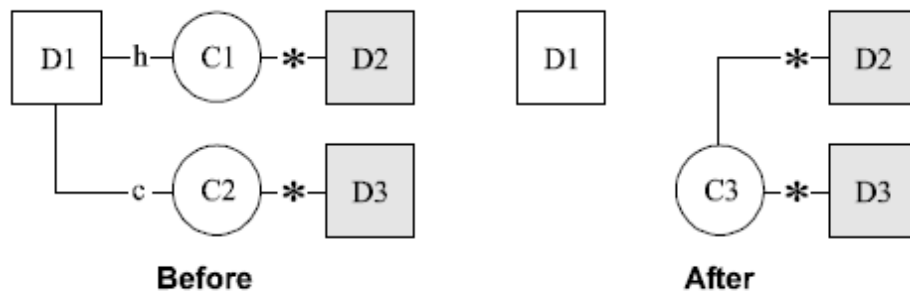
The connection being transferred may be in the Alerting, Connected, Held, or Queued state.



This service drops D1 from the call it is transferring (C1), places a new call (C3) to the transfer-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 Parameters: `universalFailure`

Functional Description:

The Alternate Call Service allows a client to put an existing active call (`activeCall`) on hold and then either 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 either

- 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, if the call is in alerting state),
- fails to connect the alerting `otherCall` (for example, if the 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:

<code>activeCall</code>	[mandatory] A valid connection identifier that indicates the <code>callID</code> and the station extension (<code>STATIC_ID</code>). The <code>deviceID</code> in <code>activeCall</code> must contain the station extension of the controlling device. The local connection state of the call can be either active or held.
<code>otherCall</code>	[mandatory] A valid connection identifier that indicates the <code>callID</code> and the station extension (<code>STATIC_ID</code>). The <code>deviceID</code> in <code>otherCall</code> must contain the station extension of the controlling device. The local connection state of the call can be alerting, bridged, or held.

Ack Parameters:

None for this service.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The error parameter in this event may contain the following error values, or one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997. • <code>INVALID_CSTA_DEVICE_IDENTIFIER</code> (12) – An invalid device identifier or extension is specified in <code>activeCall</code> or <code>otherCall</code>. • <code>INVALID_CSTA_CONNECTION_IDENTIFIER</code> (13) – An incorrect <code>callID</code>, an incorrect <code>deviceID</code>, or a dynamic device ID type is specified in <code>activeCall</code> or <code>otherCall</code>. • <code>GENERIC_STATE_INCOMPATIBILITY</code> (21) – The <code>otherCall</code> station user did not go off-hook within five seconds and is not capable of being forced off-hook. • <code>INVALID_OBJECT_STATE</code> (22) – The <code>otherCall</code> is not in the alerting, connected, held, or bridged state. • <code>INVALID_CONNECTION_ID_FOR_ACTIVE_CALL</code> (23) – The controlling <code>deviceID</code> in <code>activeCall</code> and <code>otherCall</code> is different. • <code>NO_ACTIVE_CALL</code> (24) – The <code>activeCall</code> 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.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `activeCall` and `otherCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- `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;
} CSTAEvent_t;

typedef struct CSTAAlternateCallConfEvent_t {
    Nulltype          null;
} CSTAAlternateCallConfEvent_t;

```


Answer Call Service

Summary

- Direction: Client to Switch
- Function: `cstaAnswerCall()`
- Private Data Function: `attAnswerCall`(Private data 15 and later)
- Confirmation Event: `CSTAAnswerCallConfEvent`
- Service Parameters: `alertingCall`
- Private Parameters: `videoCall`
- Ack Parameters: `noData`
- Nak Parameters: `universalFailure`

Functional Description

The Answer Call Service allows a client application to request 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 headset 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. Typically, the application will obtain the connection identifier of the alerting call from a Delivered Event Report received by the application prior to this making request.

The Answer Call Service can be used to answer a call present at any station type (for example, analog, DCP, H.323, and SIP).

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`).

Private Parameters:

`videoCall` [optional] contains Call Type. This parameter used to convey to end points media type of the call .This parameter only applies to the `attAnswerCall()` private data formatting function, which is available beginning with private data version 15.

The following media type are supported

- `VID_DISABLE` (Default) – media type as Audio
- `VID_INCOMING` – media type as audio irrespective of capabilities of end point. Will receive video of other party/call initiate party but party answer call does not send video
- `VID_TWOWAY` – media type as Video. Both party will support as video.

Ack Parameters:s

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `alertingCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- `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;
} CSTAEvent_t;

typedef struct CSTAAnswerCallConfEvent_t {
    Nulltype          null;
} CSTAAnswerCallConfEvent_t;

```

Private Data Version 15 and Later Syntax

```

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

/*
 * attAnswerCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attAnswerCall(
    ATTPriData_t      *privateData,
    ATTVideoCall_t    videocall);

/* specified */

typedef struct ATTPriData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPriData_t;

typedef enum ATTVideoCall_t{
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2
} ATTVideoCall_t;

```

Clear Call Service

Summary

- Direction: Client to Switch
- Function: `cstaClearCall()`
- Confirmation Event: `CSTAClearCallConfEvent`
- Service Parameters: `call`
- Ack Parameters: `noData`
- Nak Parameters: `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 Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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
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;
} 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 Functions: `attV6ClearConnection()` (private data versions 6 and later), `attClearConnection()` (private data versions 2 and later)
- Service Parameters: `call`
- Private Parameters: `dropResource`, `userInfo`
- Ack Parameters: `noData`
- Nak Parameters: `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

<code>call</code>	[mandatory] A valid connection identifier that indicates the endpoint to be disconnected.

Private Parameters:

<code>dropResource</code>	[optional] Specifies the resource to be dropped from the call. The available resources are <code>DR_CALL_CLASSIFIER</code> and <code>DR_TONE_GENERATOR</code> . The tone generator is any Communication Manager applied denial tone that is timed by the switch.
<code>userInfo</code>	[optional] Contains user-to-user information. This parameter allows an application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) 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 <code>NULL</code> indicates this parameter is not present.

**NOTE:**

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED (0)` – The specified data provided for the `userInfo` parameter exceeds the maximum allowable size. For private data versions 2-5, the maximum length of `userInfo` is 32 bytes. Beginning with private data version 6, 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 AE Services servers 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;
} CSTAEvent_t;

typedef struct CSTAClearConnectionConfEvent_t {
    Nulltype        null;
} CSTAClearConnectionConfEvent_t;

```

Private Data Version 6 and Later Syntax

```

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

/*
 * attV6ClearConnection() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV6ClearConnection(
    ATTPrivateData_t      *privateData,
    ATTDropResource_t     dropResource, /* DR_NONE
indicates no

                                * dropResource specified */
    ATTUserToUserInfo_t   *userInfo); /* NULL
indicates no userInfo

                                * specified */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short                length; /* 0
indicates no UUI */
        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
 * Formatting Function
 */

RetCode_t attClearConnection(
    ATTPrivateData_t      *privateData,
    ATTDropResource_t     dropResource,    /* DR_NONE
indicates no

                                * dropResource specified */
    ATTV5UserToUserInfo_t *userInfo);      /* NULL indicates no

                                * userInfo specified */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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 struct ATTV5UserToUserInfo_t {
    ATTUIProtocolType_t type;
    struct
    {
        short                length;    /* 0
indicates no UUI */
        unsigned char        value[33];
    } data;
} ATTV5UserToUserInfo_t;

typedef enum ATTUIProtocolType_t {
    UUI_NONE = -1,                                /* indicates not
specified */
    UUI_USER_SPECIFIC = 0,                        /* user-specific */

```



```
        UUI_IA5_ASCII = 4  
ASCII
```

```
/* null-terminated
```

```
*/
```

```
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 and later)
- Service Parameters: `heldCall`, `activeCall`
- Ack Parameters: `newCall`, `connList`
- Ack Private Parameters: `ucid`
- Nak Parameters: `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:

<code>heldCall</code>	[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 <code>activeCall</code> . The <code>deviceID</code> in <code>heldCall</code> must contain the station extension of the controlling device.
<code>activeCall</code>	[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 <code>heldCall</code> . The <code>deviceID</code> in <code>activeCall</code> must contain the station extension of the controlling device.

Ack Parameters:

<code>newCall</code>	[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.
<code>connList</code>	[optional – supported] Specifies the devices on the resulting <code>newCall</code>. This includes a count of the number of devices in the new call and a list of up to six <code>connectionIDs</code> and up to six <code>deviceIDs</code> 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 <code>deviceID</code> 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:

<code>ucid</code>	[optional] Specifies the Universal Call ID (UCID) of <code>newCall</code> . 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 <code>ucid</code> contains the <code>ATT_NULL_UCID</code> (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The <code>error</code> parameter in this event may contain the following error values, or one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997. • <code>INVALID_CSTA_DEVICE_IDENTIFIER</code> (12) – An invalid device identifier or extension is specified in <code>heldCall</code> or <code>activeCall</code>. • <code>INVALID_CSTA_CONNECTION_IDENTIFIER</code> (13) – The controlling <code>deviceID</code> in <code>heldCall</code> or <code>activeCall</code> has not been specified correctly.

- `GENERIC_STATE_INCOMPATIBILITY` (21) – Both calls are alerting, 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 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 AE Services servers 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 or thirteenth party to an existing twelve-party conference call (for ASAI link version 11 and above and 12 party conference enabled on CM). 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. Similarly, if a station places a twelve-party conference call on hold and another party adds yet another station (so that there are again twelve 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.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `activeCall` and `heldCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- `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;
} CSTAEvent_t;

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

typedef ExtendedDeviceID_t SubjectDeviceID_t;

```

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

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

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

Private Data Version 5 and Later Syntax

```

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

/*
 * ATTConferenceCallConfEvent - Service Response Private Data
 * (for private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_CONFERENCE_CALL_CONF */
    union
    {
        ATTConferenceCallConfEvent_t    conferenceCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTConferenceCallConfEvent_t
{
    ATTUCID_t        ucid;
} ATTConferenceCallConfEvent_t;

```


Consultation Call Service

Summary

- Direction: Client to Switch
- Function: `cstaConsultationCall()`
- Confirmation Event: `CSTAConsultationCallConfEvent`
- Private Data Function: `attV15ConsultationCall()` (private data version 15 and later), `attV10ConsultationCall()` (private data version 10-14), `attV6ConsultationCall()` (private data version 6 and later), `attConsultationCall()` (private data version 2 and later)
- Private Data Confirmation Event: `ATTConsultationCallConfEvent` (private data version 5 and later)
- Service Parameters: `activeCall`, `calledDevice`
- Private Parameters: `destRoute`, `priorityCalling`, `userInfo`, `consultOptions`, `videoCall`
- Ack Parameters: `newCall`
- Ack Private Parameters: `ucid`
- Nak Parameters: `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 (OCI) 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:

<code>activeCall</code>	[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 <code>activeCall</code> must be a station. If the party specified in the request refers to a trunk device, the request will be denied. The <code>deviceID</code> in <code>activeCall</code> must contain the station extension of the controlling device.
<code>calledDevice</code>	[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 <code>calledDevice</code> 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 <code>calledDevice</code> as if they were entered from the voice terminal using the keypad.

Private Parameters:

<code>destRoute</code>	[optional] Specifies the TAC/ARS/AAR information for an off-PBX destination, if the information is not included in the <code>calledDevice</code> . A <code>NULL</code> indicates this parameter is not specified. A TAC should not be used when the trunk is an ISDN trunk.
<code>priority-Calling</code>	[mandatory] Specifies the priority of the call. Values are On (<code>TRUE</code>) or Off (<code>FALSE</code>). If On is selected, a priority call is attempted for an on-PBX <code>calledDevice</code> . 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 (private data versions 2-5) or 96 (private data versions 6 and later) 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.

**NOTE:**

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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.

`consultOptions`

[mandatory] Specifies the intended purpose of the consultation call. This parameter only applies to the `attV10ConsultationCall()` private data formatting function, which is available beginning with private data version 10.

Valid values are:

- `CO_CONSULT_ONLY` – Indicates that the intended purpose of the consultation call is to consult with the called party.
- `CO_TRANSFER_ONLY` – Indicates that the intended purpose of the consultation call is to set up a transfer.
- `CO_CONFERENCE_ONLY` – Indicates that the intended purpose of the consultation call is to set up a conference.
- `CO_UNRESTRICTED` – Indicates that the consultation call may be used for any purpose.

The specified consult option does not actually restrict subsequent handling of the consultation call. For example, an application that has specified the consult option `CO_TRANSFER_ONLY` may use the consultation call to set up a conference rather than a transfer.

The consult option does affect the value of the private data Consult Mode field in the CSTA [Held Event](#), [Service Initiated Event](#), and [Originated Event](#) resulting from the consultation call.

`videoCall`

[optional] contains Call Type. This parameter used to convey to end points media type of the call .This parameter only applies to the `attV15ConsultationCall()` private data formatting function, which is available beginning with private data version 15.

The following media type are supported

- `VID_DISABLE` (Default) – media type as Audio
- `VID_INCOMING` – media type as audio irrespective of capabilities of end point. Will receive video of other party/call initiate party but party answering/consult call does not send video
- `VID_TWOWAY` – media type as Video. Both party will support as video.

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 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED` (0) – The specified data provided for the `userInfo` parameter exceeds the maximum allowable size. For private data versions 2-5, the maximum length of `userInfo` was 32 bytes. Beginning with private data version 6, the maximum length of `userInfo` was increased to 96 bytes. See the description of the `userInfo` parameter.
- `VALUE_OUT_OF_RANGE` (3) – The value specified for one or more of the service parameters is invalid.

- `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 AE Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, etc.) to the same device.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `activeCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- `OUTSTANDING_REQUEST_LIMIT_EXCEEDED` (44) – The client attempted to put a third party (two parties are on hold already) on hold at 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](#) in the “Make Call Service” section.

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;
} CSTAEvent_t;

typedef struct CSTAConsultationCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAConsultationCallConfEvent_t;

```

Private Data Version 15 and Later Syntax

```

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

/*
 * attV15ConsultationCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV15ConsultationCall(
    ATTPrivateData_t *privateData,
    DeviceID_t *destRoute, /* NULL
indicates not

                                * specified */
    Boolean priorityCalling, /* TRUE =
On, FALSE = Off */
    ATTUserToUserInfo_t *userInfo, /* NULL
indicates no

                                * userInfo specified */
    ATTConsultOptions_t consultOptions,
    ATTVideoCall_t videocall /*Video Option */);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        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 ATTConsultOptions_t {
    CO_UNRESTRICTED = 0,
    CO_CONSULT_ONLY = 1,
    CO_TRANSFER_ONLY = 2,
    CO_CONFERENCE_ONLY = 3
} ATTConsultOptions_t;

typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2,
} ATTVideoCall_t;

/*
 * ATTConsultationCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_CONSULTATION_CALL_CONF */
    union
    {
        ATTConsultationCallConfEvent_t    consultationCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTConsultationCallConfEvent_t
{
    ATTUCID_t    ucid;
} ATTConsultationCallConfEvent_t;

```


Private Data Version 10-14 Syntax

```

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

/*
 * attV10ConsultationCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV10ConsultationCall(
    ATTPrivateData_t *privateData,
    DeviceID_t *destRoute, /* NULL
indicates not

                                * specified */
    Boolean priorityCalling, /* TRUE =
On, FALSE = Off */
    ATTUserToUserInfo_t *userInfo, /* NULL
indicates no

                                * userInfo specified */
    ATTConsultOptions_t consultOptions);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        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 ATTConsultOptions_t {
    CO_UNRESTRICTED = 0,
    CO_CONSULT_ONLY = 1,
    CO_TRANSFER_ONLY = 2,
    CO_CONFERENCE_ONLY = 3
} ATTConsultOptions_t;

/*
 * ATTConsultationCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_CONSULTATION_CALL_CONF */
    union
    {
        ATTConsultationCallConfEvent_t    consultationCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTConsultationCallConfEvent_t
{
    ATTUCID_t        ucid;
} ATTConsultationCallConfEvent_t;

```

Private Data Version 6-9 Syntax

```

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

/*
 * attV6ConsultationCall() - Service Request Private Data
 * Formatting 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 no

                                * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        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;

/*
 * ATTConsultationCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_CONSULTATION_CALL_CONF */
    union
    {
        ATTConsultationCallConfEvent_t    consultationCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTConsultationCallConfEvent_t
{
    ATTUCID_t    ucid;
} ATTConsultationCallConfEvent_t;

```

Private Data Version 2-5 Syntax

```

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

/*
 * attConsultationCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attConsultationCall(
    ATTPrivateData_t *privateData,
    DeviceID_t *destRoute, /* NULL
indicates not

        * specified */
    Boolean priorityCalling; /* TRUE =
On, FALSE = Off */
    ATTV5UserToUserInfo_t *userInfo); /* NULL indicates no

        * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        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
* (private data version 5 and later)
*/

typedef struct
{
    ATTEventType_t    eventType; /* ATT_CONSULTATION_CALL_CONF */
    union
    {
        ATTConsultationCallConfEvent_t    consultationCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTConsultationCallConfEvent_t
{
    ATTUCID_t    ucid;
} ATTConsultationCallConfEvent_t;

```

Consultation Direct-Agent Call Service

Summary

- Direction: Client to Switch
- Function: `cstaConsultationCall()`
- Confirmation Event: `CSTAConsultationCallConfEvent`
- Private Data Function: `attV6DirectAgentCall()` (private data version 6 and later), `attDirectAgentCall()` (private data version 2 and later)
- Private Data Confirmation Event: `ATTConsultationCallConfEvent` (private data version 5 and later)
- Service Parameters: `activeCall`, `calledDevice`
- Private Parameters: `split`, `priorityCalling`, `userInfo`
- Ack Parameters: `newCall`
- Ack Private Parameters: `ucid`
- Nak Parameters: `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 the Make Direct-Agent Call Service.

Like the Consultation Call Service, the Consultation Direct-Agent Call service has the important special property of associating the Communication Manager Original Call Information (OCI) 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 Expert Agent Selection (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 Communication Manager 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 Communication Manager for that agent.

Service Parameters:

<code>activeCall</code>	[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 <code>activeCall</code> must be a station. If the party specified in the request refers to a trunk device, the request will be denied. The <code>deviceId</code> in <code>activeCall</code> must contain the station extension of the controlling device.
<code>calledDevice</code>	[mandatory] Must be a valid ACD agent extension. The agent at <code>calledDevice</code> must be logged in.

Private Parameters:

<code>Split</code>	[mandatory] Contains a valid split extension. The agent at <code>calledDevice</code> must be logged into this split.
<code>priorityCalling</code>	[mandatory] Specifies the priority of the call. Values are On (<code>TRUE</code>) or Off (<code>FALSE</code>). If On is selected, a priority call is attempted for an on-PBX <code>calledDevice</code> . Note that Communication Manager does not permit priority calls to certain types of extensions (such as VDNs).
<code>userInfo</code>	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) 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 <code>CSTARouteRequestEvent</code> to the application. A <code>NULL</code> indicates this parameter is not present.



NOTE:

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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:

<code>newCall</code>	[mandatory] A connection identifier indicates the connection between the controlling device and the new call. The <code>newCall</code> parameter contains the <code>callID</code> 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 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED (0)` – The specified data provided for the `userInfo` parameter exceeds the maximum allowable size. For private data versions 2-5, the maximum length of `userInfo` was 32 bytes. Beginning with private data version 6, the maximum length of `userInfo` was increased to 96 bytes. See [userInfo](#) on page 299.
- `GENERIC_UNSPECIFIED (0) (CS3/11, CS3/15)` – Agent is not a member of the split or 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 were detected.
- `INVALID_CALLING_DEVICE (5) (CS3/27)` – The `callingDevice` is out of service or not administered correctly on the switch.
- `PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS0/21, CS0/52)` – The COR check for completing the call failed, or 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 AE Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) for the same device.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) (CS0/50) – A service or option required for this request is not subscribed or provisioned. The request failed for one of the following reasons:
 - Answering Machine Detection must be enabled.
 - The device identifier specified in `activeCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- `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](#) 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;
} CSTAEvent_t;

typedef struct CSTAConsultationCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAConsultationCallConfEvent_t;

```

Private Data Version 6 and Later Syntax

```

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

/*
 * attV6DirectAgentCall() - Service Request Private Data
 * Formatting 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 no

                                * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        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;

/*
 * ATTConsultationCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_CONSULTATION_CALL_CONF */
    union
    {
        ATTConsultationCallConfEvent_t    consultationCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTConsultationCallConfEvent_t
{
    ATTUCID_t    ucid;
} ATTConsultationCallConfEvent_t;

```

Private Data Version 2-5 Syntax

```

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

/*
 * attDirectAgentCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attDirectAgentCall(
    ATTPrivateData_t *privateData,
    DeviceID_t *split, /*
NULL indicates not

        * specified */
    Boolean priorityCalling, /* TRUE =
On, FALSE = Off */
    ATTV5UserToUserInfo_t *userInfo); /* NULL indicates no

        * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        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
* (private data version 5 and later)
*/

typedef struct
{
    ATTEventType_t    eventType; /* ATT_CONSULTATION_CALL_CONF */
    union
    {
        ATTConsultationCallConfEvent_t    consultationCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTConsultationCallConfEvent_t
{
    ATTUCID_t    ucid;
} ATTConsultationCallConfEvent_t;

```


Consultation Supervisor-Assist Call Service

Summary

- Direction: Client to Switch
- Function: `cstaConsultationCall()`
- Confirmation Event: `CSTAConsultationCallConfEvent`
- Private Data Function: `attV6SupervisorAssistCall()` (private data version 6 and later), `attSupervisorAssistCall()` (private data version 2 and later)
- Private Data Confirmation Event: `ATTConsultationCallConfEvent`
- Service Parameters: `activeCall`, `calledDevice`
- Private Parameters: `split`, `userInfo`
- Ack Parameters: `newCall`
- Ack Private Parameters: `ucid`
- Nak Parameters: `universalFailure`

Functional Description:

The Consultation Supervisor-Assist Call Service places an existing active call (`activeCall`) on hold at an ACD agent's extension 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 the Make Supervisor-Assist Call Service.

Like the Consultation Call Service, the Consultation Supervisor-Assist Call service has the important special property of associating the Communication Manager Original Call Information (OCI) 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 active call 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 supervisor-assist 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:

<code>activeCall</code>	[mandatory] A valid connection identifier that indicates the connection to be placed on hold. This connection must be in the active (talking) state or already held. The device associated with the <code>activeCall</code> must be a valid ACD agent extension. The agent must be logged in. If the party specified in the request refers to a trunk device, the request will be denied. The <code>deviceID</code> in <code>activeCall</code> must contain the station extension of the controlling device.
<code>calledDevice</code>	[mandatory] Must be a valid on-PBX station extension (excluding VDNs, splits, off-PBX DCS and UDP extensions). The agent at <code>calledDevice</code> must be logged in.

Private Parameters:

<code>split</code>	[mandatory] Specifies the ACD agent's split extension. The ACD agent device associated with <code>activeCall</code> must be logged into this split.
<code>userInfo</code>	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) 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 <code>CSTARouteRequestEvent</code> to the application. A <code>NULL</code> indicates this parameter is not present.

NOTE:

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED (0)` – The specified data provided for the `userInfo` parameter exceeds the maximum allowable size. For private data versions 2-5, the maximum length of `userInfo` was 32 bytes. Beginning with private data version 6, the maximum length of `userInfo` was increased to 96 bytes. See the description of the [userInfo](#) parameter on page 308.
- `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 on 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 AE Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) to the same device.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) (CS0/50) – A service or option required for the request is not subscribed or provisioned. The request failed for one of the following reasons:
 - Answering Machine Detection must be enabled.
 - The device identifier specified in `activeCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- `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 [Detailed Information](#) in the "Make Supervisor-Assist 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;
} CSTAEvent_t;

typedef struct CSTAConsultationCallConfEvent_t {
    ConnectionID_t newCall;
} CSTAConsultationCallConfEvent_t;

```

Private Data Version 6 and Later Syntax

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

/*
 * attV6SupervisorAssistCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV6SupervisorAssistCall(
    ATTPrivateData_t      *privateData,
    DeviceID_t            *split,          /* mandatory.

    * NULL indicates not

    * specified */
    ATTUserToUserInfo_t   *userInfo); /* NULL indicates no

    * userInfo specified */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO      129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t   type;
    struct
    {
        short                length;          /* 0
indicates no UUI */
        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;
```

```

/*
 * ATTConsultationCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_CONSULTATION_CALL_CONF */
    union
    {
        ATTConsultationCallConfEvent_t    consultationCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTConsultationCallConfEvent_t {
    ATTUCID_t    ucid;
} ATTConsultationCallConfEvent_t;

```


Private Data Version 2-5 Syntax

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

/*
 * attSupervisorAssistCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attSupervisorAssistCall(
    ATTPrivateData_t *privateData,
    DeviceID_t *split, /*
mandatory.

    * NULL indicates not
    * specified */
    ATTV5UserToUserInfo_t *userInfo); /* NULL indicates no
    * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct {
        short length; /* 0
indicates no UUI */
        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
* (private data version 5 and later)
*/

typedef struct
{
    ATTEventType_t    eventType; /* ATT_CONSULTATION_CALL_CONF */
    union
    {
        ATTConsultationCallConfEvent_t    consultationCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTConsultationCallConfEvent_t {
    ATTUCID_t    ucid;
} ATTConsultationCallConfEvent_t;

```

Deflect Call Service

Summary

- Direction: Client to Switch
- Function: `cstaDeflectCall()`
- Confirmation Event: `CSTADeflectCallConfEvent`
- Service Parameters: `deflectCall`, `calledDevice`
- Ack Parameters: `noData`
- Nak Parameters: `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:

<code>deflectCall</code>	[mandatory] Specifies the <code>connectionID</code> 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.
<code>calledDevice</code>	[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 <code>calledDevice</code> may be stations, queues, announcements, VDNs, or logical agent extensions.

Ack Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `PRIVILEGE_VIOLATION_ON_CALLED_DEVICE` (9) (CS3/42)

The request has failed for one of the following reasons:

- An attempt was made to deflect a call back to the call originator or to the deflecting device itself.
- An attempt was made to deflect a call on the `calledDevice` of a `cstaMakePredictiveCall()` request.
- The `calledDevice` is an external number, and trunk group administration for the selected trunk group is incompatible with the request. (For example, Disconnect Supervision for outgoing calls is not enabled for the trunk group.)

- `INVALID_OBJECT_STATE` (22) (3/63)

The request has failed for one of the following reasons:

- An invalid `callID` or device identifier is specified in `deflectCall`.
- The `deflectCall` is not in alerting state.
- Attempted to deflect the call while it is in vector processing.

- `PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE` (8) (CS3/43)

The request has failed for one of the following reasons:

- An invalid `calledDevice` was specified.
- There are toll restrictions on the `calledDevice`.
- There are COR restrictions on the `calledDevice`.
- The `calledDevice` is a remote access extension.
- There is a call origination restriction on the deflecting device.
- The 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)

The request has failed for one of the following reasons:

- The service was requested on a Communication Manager administered with a release earlier than G3V4.
- The device identifier in `deflectCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- `GENERIC_OPERATION (1)` (CS0/111) – This service was requested on a queued call or there was a 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 the `cstaMonitor-Device()` service or the call is monitored by the `cstaMonitorCallsVia-Device()` service, the monitor will receive a Diverted Event when the call is successfully redirected, but there will be no Diverted Event for a monitor created with the `cstaMonitorCall()` service.
- 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`.

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;
} 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 Parameters: `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 performed a switch-hook flash on an analog station.

Service Parameters:

<code>activeCall</code>	[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 <code>activeCall</code> must be a station. If the party specified in the request refers to a trunk device, the request will be denied. The <code>deviceId</code> in <code>activeCall</code> must contain the station extension of the controlling device.
<code>reservation</code>	[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 <code>TRUE</code> . In actuality, the TSAPI Service ignores the application-supplied value for this parameter.

Ack Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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, it is in the 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 AEI Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, etc.) for the same device.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `activeCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- `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 cannot switch between a soft-held call and an active call from the voice set. However, with the Hold Call Service, this is possible by placing the active call on hard-hold and retrieving the soft-held call. The 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 the 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 – The 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 TRUE */
    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;
} 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: `attV15MakeCall()` (private data version 15 and later), `attV6MakeCall()` (private data version 6-14), `attMakeCall()` (private data version 2 and later)
- Private Data Confirmation Event: `ATTMakeCallConfEvent` (private data version 5 and later)
- Service Parameters: `callingDevice`, `calledDevice`
- Private Parameters: `destRoute`, `priorityCalling`, `userInfo`, `videoCall`
- Ack Parameters: `newCall`
- Ack Private Parameters: `ucid`
- Nak Parameters: `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 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` 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, 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:

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 events 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. An 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:

<code>destRoute</code>	[optional] Specifies the TAC/ARS/AAR information for an off-PBX destination, if the information is not included in the <code>calledDevice</code> . A <code>NULL</code> indicates that this parameter is not specified. A TAC should not be used when the trunk is an ISDN trunk.
<code>priorityCalling</code>	[mandatory] Specifies the priority of the call. Values are "On" (<code>TRUE</code>) or "Off" (<code>FALSE</code>). If On is selected, a priority call is attempted for an on-PBX <code>calledDevice</code> . Note that Communication Manager does not permit priority calls to certain types of extensions (such as VDNs).
<code>userInfo</code>	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) 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 <code>CSTARouteRequestEvent</code> to the application. A <code>NULL</code> indicates this parameter is not present.

**NOTE:**

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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.

`videoCall`

[optional] contains Call Type. This parameter used to convey to end points media type of the call .This parameter only applies to the `attV15MakeCall()` private data formatting function, which is available beginning with private data version 15.

The following media type are supported

- `VID_DISABLE` (Default) – media type as Audio
- `VID_INCOMING` – media type as audio irrespective of capabilities of end point. Will receive video of other party/call initiate party but party make call does not send video
- `VID_TWOWAY` – media type as Video. Both party will support as video.

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 Parameters:

`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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED (0)` – The specified data provided for the `userInfo` parameter exceeds the maximum allowable size. For private data versions 2-5, the maximum length of `userInfo` is 32 bytes. Beginning with private data version 6, the maximum length of `userInfo` is 96 bytes. See the description of the [userInfo](#) parameter on page 328.

- `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 AE Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) for the same device.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `callingDevice` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

- 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 – The 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 active²) 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.

² 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.

- **Attendants** – The attendant group is not supported for the 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** – The 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** – The Make Call Service is permitted to follow the VDN in the coverage path, provided that the coverage criteria have been met.
- **Call Destination** – If the `calledDevice` is an 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.

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 request 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 the 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 – A Make Call Service request to trunk groups with the Forced Entry of Account Codes feature assigned is allowed. It is up to the user at the `callingDevice` to enter the account codes via the touch-tone pad. The account code may not be provided via 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 becomes 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 – A Make Call Service request to a split 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 the Make Call Service.
- Send All Calls (SAC) – The 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 – The Make Service request accepts single-digit dialing (for example, 0 for operator).
- Skill Hunt Groups – The 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 – A VDN cannot be the `callingDevice` of a Make Call Service request, 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_MAKE_CALL_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAMakeCallConfEvent_t makeCall;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAMakeCallConfEvent_t {
    Nulltype      null;
} CSTAMakeCallConfEvent_t;

```

Private Data Version 15 and Later Syntax

```

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

/(
 * attV15MakeCall() - Service Request Private Data Formatting Function
 */

RetCode_t attV15MakeCall(
    ATTPrivateData_t *privateData,
    DeviceID_t *destRoute, /* NULL
indicates not
                                * specified */
    Boolean priorityCalling, /* TRUE =
On, FALSE = Off */
    ATTUserToUserInfo_t *userInfo, /* NULL indicates no
    ATTVideoCall_t videoCall);
                                * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        unsigned char value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2,
} ATTVideoCall_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;

/*
 * ATMakeCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType;  /* ATT_MAKE_CALL_CONF */
    union
    {
        ATMakeCallConfEvent_t    makeCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATMakeCallConfEvent_t {
    ATTUCID_t        ucid;
} ATMakeCallConfEvent_t;

```

Private Data Version 6 - 14 Syntax

```

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

/(
 * attV6MakeCall() - Service Request Private Data Formatting 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 no

                                * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        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;

```



```

/*
 * ATMakeCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_MAKE_CALL_CONF */
    union
    {
        ATMakeCallConfEvent_t    makeCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATMakeCallConfEvent_t {
    ATTUCID_t        ucid;
} ATMakeCallConfEvent_t;

```

Private Data Version 2-5 Syntax

```

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

/* attMakeCall() - Service Request Private Data Formatting Function */

RetCode_t attMakeCall(
    ATTPrivateData_t      *privateData,
    DeviceID_t            *destRoute,          /* NULL
indicates not

                                * specified */
    Boolean                priorityCalling,    /* TRUE =
On, FALSE = Off */
    ATTV5UserToUserInfo_t *userInfo);          /* NULL indicates no

                                * userInfo specified */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short          length;          /* 0
indicates no UUI */
        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
 * (private data version 5 and later)
 */

```

```
typedef struct
{
    ATTEventType_t    eventType; /* ATT_MAKE_CALL_CONF */
    union
    {
        ATTMakeCallConfEvent_t  makeCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTMakeCallConfEvent_t {
    ATTUCID_t      ucid;
} ATTMakeCallConfEvent_t;
```

Make Direct-Agent Call Service

Summary

- Direction: Client to Switch
- Function: `cstaMakeCall()`
- Confirmation Event: `CSTAMakeCallConfEvent`
- Private Data Function: `attV6DirectAgentCall()` (private data version 6 and later), `attDirectAgentCall()` (private data version 2 and later)
- Private Data Confirmation Event: `ATTMakeCallConfEvent` (private data version 5 and later)
- Service Parameters: `callingDevice`, `calledDevice`
- Private Parameters: `split`, `priorityCalling`, `userInfo`
- Ack Parameters: `newCall`
- Ack Private Parameters: `ucid`
- Nak Parameters: `universalFailure`

Functional Description:

The Make Direct-Agent Call Service is a special variation of the Make 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. The service attempts to create a new call and establish a connection with the originating device (`callingDevice`). 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:**

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. The agent at `calledDevice` must be logged in. If `calledDevice` is a logical agent's ID, it is already treated by Communication Manager 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:

<code>split</code>	[mandatory] Contains a valid split extension. The agent at <code>calledDevice</code> must be logged into this split.
<code>priorityCalling</code>	[mandatory] Specifies the priority of the call. Values are On (<code>TRUE</code>) or Off (<code>FALSE</code>). If On is selected, a priority call is attempted for an on-PBX <code>calledDevice</code> . Note that Communication Manager does not permit priority calls to certain types of extensions (such as VDNs).
<code>userInfo</code>	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) 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 <code>NULL</code> indicates that this parameter is not present.

**NOTE:**

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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:

<code>newCall</code>	[mandatory] A connection identifier that indicates the connection between the originating device and the call. The <code>newCall</code> parameter contains the <code>callID</code> of the call and the station extension of the <code>callingDevice</code> .

Ack Private Parameters:

<code>ucid</code>	[optional] Specifies the Universal Call ID (UCID) of <code>newCall</code> . 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 <code>ucid</code> contains the <code>ATT_NULL_UCID</code> (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

Nak Parameters:`universalFailure`

A Make Direct-Agent 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED (0)` – The specified data provided for the `userInfo` parameter exceeds the maximum allowable size. For private data versions 2-5, the maximum length of `userInfo` was 32 bytes. Beginning with private data version 6, the maximum length of `userInfo` is 96 bytes. See the description of the [userInfo](#) parameter on page 344.
- `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 were detected.
- `INVALID_CALLING_DEVICE (5) (CS3/27)` – The `callingDevice` is out of service or not administered correctly on the switch.
- `PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS0/21, CS0/52)` – The COR check for completing the call failed, or 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)` – The request is incompatible with call state.
- `INVALID_CSTA_DEVICE_IDENTIFIER (12) (CS0/28)` – The split does not contain a valid hunt group extension, or the `callingDevice` or `calledDevice` is an invalid station extension.

- `INVALID_OBJECT_TYPE` (18) (CS0/58, CS3/80)

The request has failed for one of the following reasons:

- 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 AE Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) for the same device.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) (CS0/50) – A required service or option is not subscribed or provisioned. The request failed for one of the following reasons:
 - Answering Machine Detection must be enabled.
 - The device identifier specified in `callingDevice` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

See also, [Detailed Information](#) 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`.
- SIP Stations – When the `callingDevice` is a SIP station, Avaya Communication Manager places the call directly to the logical agent ID associated with the `calledDevice`, rather than placing the call to the `calledDevice`. This changes how the `calledDevice` parameter is set in the subsequent CSTA Delivered and CSTA Established events and, consequently, how the Original Call Information is set in those events.

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;
} CSTAEvent_t;

typedef struct CSTAMakeCallConfEvent_t {
    Nulltype      null;
} CSTAMakeCallConfEvent_t;

```

Private Data Version 6 and Later Syntax

```

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

/*
 * attV6DirectAgentCall() - Service Request Private Data
 * Formatting 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 no

                                * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        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;

/*
 * ATTMakeCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_MAKE_CALL_CONF */
    union
    {
        ATTMakeCallConfEvent_t    makeCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTMakeCallConfEvent_t
{
    ATTUCID_t            ucid;
} ATTMakeCallConfEvent_t;

```

Private Data Version 2-5 Syntax

```

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

/*
 * attDirectAgentCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attDirectAgentCall(
    ATTPrivateData_t *privateData,
    DeviceID_t *split, /*
    NULL indicates not
                        * specified */
    Boolean priorityCalling, /* TRUE =
On, FALSE = Off */
    ATTV5UserToUserInfo_t *userInfo); /* NULL indicates no
                                      * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct {
        short length; /* 0
indicates no UUI */
        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
* (private data version 5 and later)
*/

typedef struct
{
    ATTEventType_t    eventType; /* ATT_MAKE_CALL_CONF */
    union
    {
        ATTMakeCallConfEvent_t  makeCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTMakeCallConfEvent_t
{
    ATTUCID_t          ucid;
} ATTMakeCallConfEvent_t;

```

Make Predictive Call Service

Summary

- Direction: Client to Switch
- Function: `cstaMakePredictiveCall()`
- Confirmation Event: `CSTAMakePredictiveCallConfEvent`
- Private Data Function: `attV6MakePredictiveCall()` (private data version 6 and later), `attMakePredictiveCall()` (private data version 2 and later)
- Private Data Confirmation Event: `ATTMakePredictiveCallConfEvent` (private data version 5 and later)
- Service Parameters: `callingDevice`, `calledDevice`, `allocationState`
- Private Parameters: `priorityCalling`, `maxRings`, `answerTreat`, `destRoute`, `userInfo`
- Ack Parameters: `newCall`
- Ack Private Parameters: `ucid`
- Nak Parameters: `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 Communication Manager.

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:

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. An 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` only when the `calledDevice` enters the connected state.
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.

<code>answerTreat</code>	[mandatory] Specifies the call treatment when an answering machine is detected. • <code>AT_NONE</code> – Treatment follows the switch answering machine detection administration. • <code>AT_DROP</code> – Drops the call if an answering machine is detected. • <code>AT_CONNECT</code> – Connects the call to the calling device if an answering machine is detected. • <code>AT_NO_TREATMENT</code> – Indicates that answering machine detection will not be applied to the call.
<code>destRoute</code>	[optional] Specifies the TAC/ARS/AAR information for off-PBX destinations if the information is not included in the <code>calledDevice</code>. A <code>NULL</code> indicates that this parameter is not specified. A TAC should not be used when the trunk is an ISDN trunk.
<code>userInfo</code>	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) 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 <code>UUI</code> in the <code>ISDN SETUP</code> message over the PRI trunk to establish the call. The local and the remote switch include the <code>UUI</code> in the Delivered Event Report and in the <code>CSTARouteRequestEvent</code> to the application. A <code>NULL</code> indicates that this parameter is not present.

NOTE:

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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 Parameters:`universalFailure`

A Make Predictive 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED (0)` – The specified data provided for the `userInfo` parameter exceeds the maximum allowable size. For private data versions 2-5, the maximum length of `userInfo` is 32 bytes. Beginning with private data version 6, the maximum length of `userInfo` is 96 bytes. See the description the [userInfo](#) parameter on page 354.
- `VALUE_OUT_OF_RANGE (3) (CS0/100, CS0/96)` – Invalid information element contents were detected.
- `INVALID_CALLING_DEVICE (5) (CS3/27)` – The `callingDevice` is out of service or not administered correctly on the switch.
- `PRIVILEGE_VIOLATION_ON_CALLED_DEVICE (9) (CS0/21, CS0/52)`

The request has failed for one of the following reasons:

- The service request 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)` – The request is incompatible with the call state.

- `GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31) (CS3/22)`

One of the following conditions existed when the switch attempted to make the call:

- No Call classifier available
 - No time slot available
 - No trunk available
 - Queue full
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) (CS0/50)` – A required service or option is not subscribed or provisioned. The request failed for one of the following reasons:
 - Answer Machine Detection is requested, but AMD is not enabled on the switch.
 - The device identifier specified in `callingDevice` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

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 call monitor using `cstaMonitorCall()` after the application receives a confirmation for the `cstaMakePredictiveCall()` request.
- 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:

Consider using enhanced VDN monitors for predictive calling applications. For more information, see [Monitor Calls Via Device Service](#) on page 591.

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);

typedef enum AllocationState_t {
    AS_CALL_DELIVERED = 0,          /* Not supported */
    AS_CALL_ESTABLISHED = 1
} AllocationState_t;

/* 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;
} CSTAEvent_t;

typedef struct CSTAMakePredictiveCallConfEvent_t
{
    ConnectionID_t    newCall;
} CSTAMakePredictiveCallConfEvent_t;

```

Private Data Version 6 and Later Syntax

```

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

/*
 * attV6MakePredictiveCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV6MakePredictiveCall(
    ATTPrivateData_t *privateData,
    Boolean priorityCalling, /* TRUE =
On, FALSE = Off */
    short maxRings, /*
if less than 2 or greater
                    * than 15, 10 is used */
    ATTAnswerTreat_t answerTreat, /*
AT_NO_TREATMENT, AT_NONE,
                    * AT DROP, or AT_CONNECT */
    DeviceID_t *destRoute, /* NULL
indicates not
                    * specified */
    ATTUserToUserInfo_t *userInfo); /* NULL
indicates no
                    * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO 129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct {
        short length; /* 0
indicates no UUI */
        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 */
} ATTUUIProtocolType_t;

typedef enum ATTAnswerTreat_t {
    AT_NO_TREATMENT = 0,                          /* no answering machine detection */
    AT_NONE = 1,                                  /* treatment follows switch admin.
*/
    AT_DROP = 2,                                  /* drop call if ans. mach.
detected */
    AT_CONNECT = 3                               /* connect call if ans. mach.
detected */
} ATTAnswerTreat_t;

/*
 * ATTMakePredictiveCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_MAKE_PREDICTIVE_CALL_CONF
*/
    union
    {
        ATTMakePredictiveCallConfEvent_t    makePredictiveCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTMakePredictiveCallConfEvent_t {
    ATTUCID_t    ucid;
} ATTMakePredictiveCallConfEvent_t;

```

Private Data Version 2-5 Syntax

```

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

/*
 * attMakePredictiveCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMakePredictiveCall(
    ATTPrivateData_t *privateData,
    Boolean priorityCalling, /* TRUE =
On, FALSE = Off */
    short maxRings, /*
if less than 2 or greater
                    * than 15, 10 is used */
    ATAnswerTreat_t answerTreat, /*
AT_NO_TREATMENT, AT_NONE,
                    * AT DROP, or AT_CONNECT */
    DeviceID_t *destRoute, /* NULL
indicates not
                    * specified */
    ATTV5UserToUserInfo_t *userInfo); /* NULL indicates no
                    * userInfo specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct {
        short length; /* 0
indicates no UUI */
        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 ATTAnswerTreat_t {
    AT_NO_TREATMENT = 0, /* no answering machine detection */
    AT_NONE = 1,         /* treatment follows switch admin.
*/
    AT_DROP = 2,         /* drop call if ans. mach.
detected */
    AT_CONNECT = 3       /* connect call if ans. mach.
detected */
} ATTAnswerTreat_t;

/*
 * ATMakePredictiveCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_MAKE_PREDICTIVE_CALL_CONF
*/
    union
    {
        ATMakePredictiveCallConfEvent_t    makePredictiveCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATMakePredictiveCallConfEvent_t
{
    ATTUCID_t        ucid;
} ATMakePredictiveCallConfEvent_t;

```

Make Supervisor-Assist Call Service

Summary

- Direction: Client to Switch
- Function: `cstaMakeCall()`
- Confirmation Event: `CSTAMakeCallConfEvent`
- Private Data Function: `attV6SupervisorAssistCall()` (private data version 6 and later), `attSupervisorAssistCall()` (private data version 2 and later)
- Private Data Confirmation Event: `ATTMakeCallConfEvent` (private data version 5 and later)
- Service Parameters: `callingDevice`, `calledDevice`
- Private Parameters: `split`, `userInfo`
- Ack Parameters: `newCall`
- Ack Private Parameters: `ucid`
- Nak Parameters: `universalFailure`

Functional Description:

The Make Supervisor-Assist Call Service is a special variation of the 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. 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). The agent must be logged in.

**NOTE:**

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 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 (private data versions 2-5) or 96 (private data versions 6 and later) 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.

**NOTE:**

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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 Parameters:

`universalFailure` A Make Supervisor-Assist 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED (0)` – The specified data provided for the `userInfo` parameter exceeds the maximum allowable size. For private data versions 2-5, the maximum length of `userInfo` is 32 bytes. Beginning with private data version 6, the maximum length of `userInfo` is 96 bytes. See the [userInfo](#) parameter on page 365.
- `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 were detected.
- `INVALID_CALLING_DEVICE (5) (CS3/27)` – The `callingDevice` is out of service or not administered correctly on 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 request has failed for one of the following reasons:

- 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) – The request is incompatible with the 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 AE Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Make Call, etc.) for the same device.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) (CS0/50) – A required service or option is not subscribed or provisioned. The request failed for one of the following reasons:
 - Answering Machine Detection must be enabled.
 - The device identifier specified in `callingDevice` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

See [Detailed Information](#) 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;
} CSTAEvent_t;

typedef struct CSTAMakeCallConfEvent_t {
    Nulltype        null;
} CSTAMakeCallConfEvent_t;

```

Private Data Version 6 and Later Syntax

```

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

/*
 * attV6SupervisorAssistCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV6SupervisorAssistCall(
    ATTPrivateData_t      *privateData,
    DeviceID_t            *split,      /*
mandatory.

        * NULL indicates not
        * specified */
    ATTUserToUserInfo_t   *userInfo); /* NULL indicates no
        * userInfo specified */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_USER_INFO    129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t   type;
    struct
    {
        short                length;      /* 0
indicates no UUI */
        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;

```

```

/*
 * ATMakeCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_MAKE_CALL_CONF */
    union
    {
        ATMakeCallConfEvent_t  makeCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATMakeCallConfEvent_t {
    ATTUCID_t      ucid;
} ATMakeCallConfEvent_t;

```


Private Data Version 2-5 Syntax

```

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

/*
 * attSupervisorAssistCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attSupervisorAssistCall(
    ATTPrivateData_t      *privateData,
    DeviceID_t            *split,      /*
mandatory.

    * NULL indicates not
    * specified */
    ATTV5UserToUserInfo_t *userInfo); /* NULL indicates no
    * userInfo specified */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short                length;      /* 0
indicates no UUI */
        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;

/*

```

```

* ATMakeCallConfEvent - Service Response Private Data
* (private data version 5 and later)
*/

typedef struct
{
    ATTEventType_t    eventType; /* ATT_MAKE_CALL_CONF */
    union
    {
        ATMakeCallConfEvent_t  makeCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATMakeCallConfEvent_t {
    ATTUCID_t          ucid;
} ATMakeCallConfEvent_t;

```

Pickup Call Service

Summary

- Direction: Client to Switch
- Function: `cstaPickupCall()`
- Confirmation Event: `CSTAPickupCallConfEvent`
- Service Parameters: `deflectCall`, `calledDevice`
- Ack Parameters: `noData`
- Nak Parameters: `universalFailure`

Functional Description:

This service redirects an alerting call at a device to another on-PBX device (which could be on a different switch in a DCS environment). The call at the alerting device is dropped after a successful redirection. An application may deflect an alerting call any number of times until the call is answered or is dropped by the caller.

The service request is positively acknowledged if the call has been successfully redirected to the `calledDevice`.

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:

<code>deflectCall</code>	[mandatory] Specifies the <code>connectionID</code> of the call that is to be redirected to another destination. The call must be in alerting state at the device. The device must be a valid voice station extension.
<code>calledDevice</code>	[mandatory] Specifies the destination of the call. The destination must be an on-PBX endpoint. The <code>calledDevice</code> may be stations, queues, announcements, VDNs, or logical agent extension. Note that the <code>calledDevice</code> can be a device within a DCS environment.

Ack Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.
- `PRIVILEGE_VIOLATION_ON_CALLED_DEVICE` (9) (CS3/42)

The request has failed for one of the following reasons:

- The `calledDevice` is the call originator or the alerting device itself.
- The `calledDevice` for this `cstaPickupCall()` service request was the `calledDevice` of an earlier `cstaMakePredictiveCall()` service request.
- The `calledDevice` is an external number, and trunk group administration for the selected trunk group is incompatible with the request. (For example, Disconnect Supervision for outgoing calls is not enabled for the trunk group.)
- `INVALID_OBJECT_STATE` (22) (3/63)

The request has failed for one of the following reasons:

- An invalid `callID` or device identifier is specified in `deflectCall`.
- The `deflectCall` is not at alerting state.
- The service attempted to redirect the call while in vector processing.
- `PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE` (14) (CS3/43)

The request has failed for one of the following reasons:

- An invalid `calledDevice` was specified.
- There are toll restrictions on the `calledDevice`.
- There are COR restrictions on the `calledDevice`.
- The `calledDevice` is a remote access extension.
- There is a call origination restriction on the redirecting device.
- The call is in vector processing.
- `RESOURCE_BUSY` (33) (CS0/17) – The `calledDevice` is a busy station, a station that has call forwarding active, or a TEG group with one or more busy members.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `deflectCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

- `GENERIC_OPERATION (1) (CS0/111)` – This service is requested on a queued call or there was a 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 `cstaPickupCall()` 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 Calls – Call redirection to a station with Call Forwarding/Cover All/Send All Calls 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 a 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()`, the monitor 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, and Connection Cleared Events if redirection fails.

NOTE:

If trunk-to-trunk transfer is disallowed by 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 `cstaPickupCall()`.

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;
} 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 and later), `attReconnectCall()` (private data version 2 and later)
- Service Parameters: `activeCall`, `heldCall`
- Private Parameters: `dropResource`, `userInfo`
- Ack Parameters: `noData`
- Nak Parameters: `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 the Retrieve Call Service or the 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:

<code>activeCall</code>	[mandatory] A valid connection identifier that indicates the <code>callID</code> and the station extension (<code>STATIC_ID</code>). The <code>deviceID</code> in <code>activeCall</code> must contain the station extension of the controlling device. The local connection state of the call must be connected.
<code>heldCall</code>	[mandatory] A valid connection identifier that indicates the <code>callID</code> and the station extension (<code>STATIC_ID</code>). The <code>deviceID</code> in <code>heldCall</code> must contain the station extension of the controlling device. The local connection state of the call can be alerting, bridged, or held.

Private Parameters:

<code>dropResource</code>	[optional] Specifies the resource to be dropped from the call. The available resources are <code>DR_CALL_CLASSIFIER</code> and <code>DR_TONE_GENERATOR</code> . The tone generator is any Communication Manager applied denial tone that is timed by the switch.
<code>userInfo</code>	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) 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 <code>NULL</code> indicates that this parameter is not present.

**NOTE:**

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED` (0) – The specified data provided for the `userInfo` parameter exceeds the maximum allowable size. For private data versions 2-5, the maximum length of `userInfo` is 32 bytes. Beginning with private data version 6, the maximum length of `userInfo` is 96 bytes. See the description of the [userInfo](#) parameter on page 379.
- `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 request has failed for one of the following reasons:

- The specified `activeCall` at the station is not currently in the connected 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 request has failed for one of the following reasons:

- The switch is busy with another CSTA request. This can happen when two AE Services servers are issuing requests (for example, Clear Connection, etc.) for the same device.
- The client attempted to add a seventh party to a call with six active parties.

- RESOURCE_BUSY (33) – The station is busy on a call or there are no idle appearances available.
- GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41) – The device identifier specified in `activeCall` and `heldCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- 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;
} CSTAEvent_t;

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

Private Data Version 6 and Later Syntax

```

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

/*
 * attV6ReconnectCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV6ReconnectCall(
    ATTPrivateData_t      *privateData,
    ATTDropResource_t     dropResource, /* DR_NONE
indicates

    * no dropResource

    * specified */
    ATTUserToUserInfo_t  *userInfo); /* NULL
indicates

    * no userInfo

    * specified */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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

typedef struct ATTUserToUserInfo_t {
    ATTUIProtocolType_t type;
    struct
    {
        short                length; /* 0
indicates no UII */
        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
 * Formatting Function
 */

RetCode_t attReconnectCall(
    ATTPrivateData_t      *privateData,
    ATTDropResource_t     dropResource, /* DR_NONE
indicates

    * no dropResource

    * specified */
    ATTV5UserToUserInfo_t *userInfo); /* NULL indicates

    * no userInfo

    * specified */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short 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 struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct
    {
        short length; /* 0
indicates no UUI */
        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: `CSTARRetrieveCallConfEvent`
- Service Parameters: `heldCall`
- Ack Parameters: `noData`
- Nak Parameters: `universalFailure`

Functional Description:

The Retrieve Call Service connects an on-PBX held connection.

Service Parameters:

<code>heldCall</code>	[mandatory] A valid connection identifier that indicates the endpoint to be connected. The <code>deviceId</code> in <code>heldCall</code> must contain the station extension of the endpoint.

Ack Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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, it is in the alerting state) so it cannot be retrieved.
- `RESOURCE_BUSY` (33) – The switch is busy with another CSTA request. This can happen when two AE Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Conference Call, etc.) for the same device.
- `CONFERENCE_MEMBER_LIMIT_EXCEEDED` (38) – The client attempted to add a seventh party to a six-party conference call or thirteenth party to a twelve-party conference call (for ASAI link version 11 and above and 12 party conference enabled on CM).
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `heldCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
- `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 – The 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 when a request for the Retrieve Call Service is received, the dial tone will be dropped and the user is 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;
} 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: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attSendDTMFToneExt()` (private data version 5 and later), `attSendDTMFTone()` (private data version 4 and later)
- Service Parameters: `noData`
- Private Parameters: `sender, receivers, tones, toneDuration, pauseDuration`
- Ack Parameters: `noData`
- Nak Parameters: `universalFailure`

Functional Description:

The Send DTMF Tone Service sends a sequence of DTMF tones (maximum of 32) on behalf of an on-PBX endpoint 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 Escape Service Confirmation event 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:

None for this service.

Private Parameters:

<code>sender</code>	[mandatory] Specifies the <code>connectionID</code> of the endpoint on whose behalf DTMF tones are to be sent. This <code>connectionID</code> can be for an on-PBX endpoint or an off-PBX endpoint (via trunk connection) on the call.
<code>receivers</code>	[optional – not supported] A list of up to five <code>connectionIDs</code> that can receive the DTMF tones. If this list is empty (<code>NULL</code> 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.
<code>tones</code>	[mandatory] DTMF sequence to be generated. The maximum length of the tone sequence is 32. The allowed DTMF tones are specified as a null-terminated ASCII string containing only the digits 0-9, '#' and '*'. Any other character in tones is invalid and will cause the request to be denied.
<code>toneDuration</code>	[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).
<code>pauseDuration</code>	[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 Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `VALUE_OUT_OF_RANGE` (3) (CS0/100) – The request failed for one of the following reasons:
 - The `tones` parameter has a length equal to 0 or greater than 32.
 - The `tones` parameter contains invalid characters.
 - The parameter values for either `toneDuration` or `pauseDuration` are invalid.
- `OBJECT_NOT_KNOWN` (4) (CS0/96) – A mandatory parameter value is missing.
- `INVALID_CSTA_DEVICE_IDENTIFIER` (13) (CS0/28) – An invalid `deviceID` is specified in `sender`.
- `INVALID_OBJECT_STATE` (22) (CS0/98, CS3/63) – The request failed for one of the following reasons:
 - The service was requested on a call that is currently receiving a switch-provided tone, such as dial tone, busy tone, ringback tone, intercept tone, Music-on-Hold/Delay, etc.
 - The sender does not have an active voice path to the call (e.g., the call is alerting or on hold). The call must be in an established state in order to send DTMF tones.
 - The service was requested on a call that is in vector processing.
 - The service was requested on a call that is being service observed.
- `NO_ACTIVE_CALL` (24) (CS3/86) – An invalid `callID` is specified in `sender` or `receivers`.
- `GENERIC_SYSTEM_RESOURCE_AVAILABILITY` (31) (CS3/40) – The request could not be completed due to a lack of available switch resources.

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 in the alerting or held local state cannot send DTMF tones. A receiver must have an active voice path to the sender. A receiver in the held local state will not receive the tones, although the switch will attempt to send the tones.
- 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 receivers.
- 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 tones 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.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t   *privateData);

/* CSTAEscapeSvcConfEvent - 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 and Later Syntax

```

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

/*
 * attSendDTMFToneExt() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attSendDTMFToneExt(
    ATTPrivateData_t      *privateData,
    ConnectionID_t        *sender,
    /* mandatory

    * NULL is treated as
    * not specified */
    ATTCConnIDList_t      *receivers,          /* ignored
- reserved

    * for future use.

    * Tones are sent to
    * all parties. */
    char                  *tones,
    /* mandatory

    * NULL is treated as
    * not specified */
    short                 toneDuration,
    /* tone duration */
    short                 pauseDuration);      /*
pause duration */

typedef struct ATTPrivateData_t {
    char                  vendor[32];
    unsigned short        length;
    char                  data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTCConnIDList_t
{
    unsigned int          count;
    ConnectionID_t        *pParty;
} ATTCConnIDList_t;

```

Private Data Version 4 Syntax

```

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

/*
 * attSendDTMFTone() - Service Request Private Data
 * Formatting 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.
    * Tones are sent to
    * all parties. */
    char                  *tones,
    /* mandatory

    * NULL is treated as
    * not specified */
    short                 toneDuration,
    short                 pauseDuration);
/* pause duration */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

#define ATT_MAX_RECEIVERS    5    /* Max receivers for touch tones
*/

typedef struct ATTV4ConnIDList_t
{

```

Send DTMF Tone Service (Private Data Version 4 and Later)

```
        unsigned 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: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attSelectiveListeningHold()` (private data version 5 and later)
- Service Parameters: `noData`
- Private Parameters: `subjectConnection, allParties, selectedParty`
- Ack Parameters: `noData`
- Nak Parameters: `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.

See the Selective Listening Retrieve Service to allow the listen-held party to be retrieved (i.e., to again hear the other party or parties on the call).

Service Parameters:

None for this service.

Private Parameters:

<code>subjectConnection</code>	[mandatory] Specifies the <code>connectionID</code> of the party who will not hear the voice from all other parties or a single party specified in the <code>selectedParty</code>. This <code>connectionID</code> can be an on-PBX endpoint or an off-PBX endpoint (via trunk connection) on the call.
<code>allParties</code>	[mandatory] Specifies that either all parties' or a single party's listening path is to be held from the <code>subjectConnection</code> party. TRUE – the listening paths of all parties on the call will be held from the <code>subjectConnection</code> party. This prevents the <code>subjectConnection</code> from listening to all other parties on the call. The <code>subjectConnection</code> endpoint can still talk and be heard by all other connected parties on the call. The <code>selectedParty</code> parameter is ignored. FALSE – the listening path of the <code>subjectConnection</code> party will be held from the <code>selectedParty</code> party. This prevents the <code>subjectConnection</code> from listening to all other parties on the call. The <code>subjectConnection</code> endpoint can still talk and be heard by all other connected parties on the call. The <code>selectedParty</code> parameter must be specified.
<code>selectedParty</code>	[optional] A <code>connectionID</code> whose voice will not be heard by the <code>subjectConnection</code> party. If <code>allParties</code> is FALSE, a <code>connectionID</code> must be specified. If <code>allParties</code> is TRUE, the <code>connectionID</code> in this parameter is ignored.

Ack Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `VALUE_OUT_OF_RANGE` (3) (CS0/100) – A party specified is not part of the call or is in the wrong state (e.g., a two-party call with the `selectedParty` still in the alerting state).
- `OBJECT_NOT_KNOWN` (4) (CS0/96) – A 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) – An 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 while 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 ASAI Link Plus Capabilities customer option must be enabled (set to 'y') on Communication Manager in order to use this feature.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 and Later Syntax

```

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

/*
 * attSelectiveListeningHold() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attSelectiveListeningHold(
    ATTPrivateData_t      *privateData,
    ConnectionID_t        *subjectConnection,
    Boolean               allParties,
    ConnectionID_t        *selectedParty);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTSelectiveListeningHoldConfEvent - 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: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attSelectiveListeningRetrieve()`
- Service Parameters: `noData`
- Private Parameters: `subjectConnection, allParties, selectedParty`
- Ack Parameters: `noData`
- Nak Parameters: `universalFailure`

Functional Description:

The Selective Listening Retrieve Service allows a client application to retrieve a party (`subjectConnection`) from listen-hold for another party (`selectedParty`) or for all parties that were previously being listen-held.

Service Parameters:

None for this service.

Private Parameters:

<code>subjectConnection</code>	[mandatory] Specifies the <code>connectionID</code> of the party whose listening path will be reconnected to all parties or to the party specified by <code>selectedParty</code> . This <code>connectionID</code> can be an on-PBX endpoint or an off-PBX endpoint (via trunk connection) on the call.
<code>allParties</code>	[mandatory] Specifies that either all parties' or a single party's listening path is to be reconnected from the <code>subjectConnection</code> party. TRUE – the listening paths of all parties on the call will be reconnected from the <code>subjectConnection</code> party. This allows the <code>subjectConnection</code> endpoint to be able to listen to all other parties on the call. The <code>selectedParty</code> parameter is ignored. FALSE – the listening path of the <code>subjectConnection</code> party will be reconnected from the <code>selectedParty</code> party. This allows the <code>subjectConnection</code> endpoint be able to listen to <code>selectedParty</code> party. The <code>selectedParty</code> parameter must be specified.
<code>selectedParty</code>	[optional] A <code>connectionID</code> whose listening path will be retrieved from listen-held by the <code>subjectConnection</code> party. If <code>allParties</code> is FALSE, <code>connectionIDs</code> must be specified. If <code>allParties</code> is TRUE, the <code>connectionID</code> in this parameter is ignored.

Ack Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `VALUE_OUT_OF_RANGE` (3) (CS0/100) – A party specified is not part of the call or is in the wrong state (e.g., a two-party call with the `selectedParty` still in the alerting state).
- `OBJECT_NOT_KNOWN` (4) (CS0/96) – A 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) – An 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);

/* CSTAEscapeSvcConfEvent - 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 and Later Syntax

```

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

/*
 * attSelectiveListeningRetrieve() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attSelectiveListeningRetrieve(
    ATTPrivateData_t      *privateData,
    ConnectionID_t        *subjectConnection,
    Boolean               allParties,
    ConnectionID_t        *selectedParty);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTSelectiveListeningRetrieveConfEvent - 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: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attSingleStepConferenceCall()` (private data version 5 and later)
- Private Data Confirmation Event: `ATTSingleStepConferenceCallConfEvent`
- Service Parameters: `noData`
- Private Parameters: `activeCall`, `deviceToBeJoin`, `participationType`, `alertDestination`
- Ack Parameters: `noData`
- Ack Private Parameters: `newCall`, `connList`, `ucid`
- Nak Parameters: `universalFailure`

Functional Description:

The Single Step Conference Call Service will join a new device to an existing call. This service can be repeated to make n-device conference calls (subject to switching function limits). Currently, Communication Manager supports six (6) parties on a call.

NOTE:

The Single Step Conference Call Service is not supported for ISDN BRI stations.

Service Parameters:

None for this service.

Private Parameters:

<code>activeCall</code>	[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.
<code>deviceToBeJoin</code>	[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, 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.
<code>participationType</code>	[optional] Specifies the type of participation for the added device in the resulting call. Possible values are: • <code>PT_ACTIVE</code> – the added device actively participates in the resulting conference call. The added device can listen and talk. • <code>PT_SILENT</code> – the added device can listen but cannot actively participate (cannot talk) in the resulting conferenced call. Thus the other parties on the call will be unaware that the added device has joined the call (there will be no display updates). This option is useful for applications that may desire to silently conference in devices (e.g., service observing). • If a party that was added to the call via the Single Step Conference Call Service with participation type <code>PT_SILENT</code> holds the call and then conferences in another party, the original <code>PT_SILENT</code> status of the party is negated (which means the party would then be heard by all other parties).
<code>alertDestination</code>	[optional – partially supported] Specifies whether or not the <code>deviceToBeJoin</code> is to be alerted. • <code>TRUE</code> – <code>deviceToBeJoin</code> will be alerted (with Delivered event) before it joins the call.

➤ NOTE:

The value "TRUE" 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 value "FALSE" is supported in the current release.

Ack Parameters:

None for this service.

Ack Private Parameters:

<code>newCall</code>	[mandatory] A <code>connectionID</code> specifies the <code>callID</code> and the <code>deviceID</code> of the joining device. The <code>callID</code> is the same <code>callID</code> as specified in the service request; that is, the <code>callID</code> of the resulting call is not changed.
<code>connList</code>	[optional – supported] Specifies the devices on the resulting <code>newCall</code>. This includes a count of the number of devices in the conferenced call and a list of <code>connectionIDs</code> and <code>deviceIDs</code> 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). A group extension may contain alerting or bridged extensions. • The static <code>deviceID</code> 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.
<code>ucid</code>	[optional – supported] Specifies the Universal Call ID (UCID) of <code>newCall</code> . 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 <code>ucid</code> contains the <code>ATT_NULL_UCID</code> (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `VALUE_OUT_OF_RANGE` (3) (CS0/100) – An unsupported was specified or an out-of-range value is specified for a parameter.
- `OBJECT_NOT_KNOWN` (4) (CS0/96) – A 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` was on-hook when the Single Step Conference Call service was initiated. The `deviceToBeJoin` should be in off-hook/auto-answer condition.
- `PRIVILEGE_VIOLATION_ON_SPECIFIED_DEVICE` (8) (CS3/43) – The class of restriction on `deviceToBeJoin` was 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.
- `INVALID_OBJECT_TYPE` (18) (CS0/58) – The call has conference restrictions 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 was made with option `PT_ACTIVE` while the call was in vector processing.
- `RESOURCE_BUSY` (33) (CS0/17) – The `deviceToBeJoin` is busy or not in the 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 `attSingleStepConferenceCall()` request will generate an Established Event followed by a Conferenced Event for call monitors and for device monitors 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 `attSingleStepConferenceCall()` 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 `attSingleStepConferenceCall()` request, and `EC_SILENT_MONITOR` if `PT_SILENT` was specified. The `confController` and `addedParty` parameters in the Conferenced Event have the same device ID as `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 the Single Step Conference Call 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 eliminates 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. Communication Manager cannot distinguish between recording devices and real stations, so if a recording device is left on a call with only one other party, Communication Manager will leave that call up forever, (or until one of those parties drops).
- **Drop Button and Last Added Party** – A party added by the Single Step Conference Call Service will never be considered as the "last added party" on the call. Thus, parties added through the Single Step Conference Call Service cannot be dropped by using the Drop button.
- **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_ID`.
- **Remote Agent Trunk to Trunk Conference/Transfer** – In this type of application, an incoming call from 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 the Single Step Conference Call 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 Call 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 the 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);

/* CSTAEscapeSvcConfEvent - 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 and Later Syntax

```

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

/*
 * attSingleStepConferenceCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attSingleStepConferenceCall(
    ATTPrivateData_t      *privateData,
    ConnectionID_t        *activeCall,
    /* mandatory */
    DeviceID_t            *deviceToBeJoin, /*
mandatory */
    ATTParticipationType_t participationType,
    Boolean                alertDestination);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTParticipationType_t {
    PT_ACTIVE = 1,
    PT_SILENT = 0
} ATTParticipationType_t;

/*
 * ATTSingleStepConferenceCallConfEvent - Service Response Private
Data
 */

typedef struct
{
    ATTEventType_t      eventType;
                                                                    /*
ATT_SINGLE_STEP_CONFERENCE_CALL_CONF */
    union
    {
        ATTSingleStepConferenceCallConfEvent_t
        ssconference;
    } u;
} ATTEvent_t;

typedef struct Connection_t {
    ConnectionID_t      party;

```

```

        SubjectDeviceID_t          staticDevice;
    } Connection_t;

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

typedef char ATTUCID_t[64];

typedef struct ATTSingleStepConferenceCallConfEvent_t {
    ConnectionID_t          newCall;
    ConnectionList_t        connList;
    ATTUCID_t               ucid;
} ATTSingleStepConferenceCallConfEvent_t;

```


Single Step Transfer Call (Private Data Version 8 and later)

Summary

- Direction: Client to Switch
- Function: `cstaEscapeService()`
- Confirmation Event: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attSingleStepTransferCall()`
- Private Data Confirmation Event: `ATTV15SingleStepTransferCallConfEvent` (private data version 15 and above), `ATTV9SingleStepTransferCallConfEvent` (private data version 9-14), `ATTV8SingleStepTransferCallConfEvent` (private data version 8)
- Service Parameters: `noData`
- Private Parameters: `activeCall`, `transferredTo`, `videoCall`
- Ack Parameters: `noData`
- Ack Private Parameters: `transferredCall`, `ucid` (private data version 9 and later)
- Nak Parameters: `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 request.

The connection being transferred may be in the Alerting, Connected, Held, or Queued state.

Service Parameters:

None for this service.

Private Parameters:

<code>activeCall</code>	[mandatory] A pointer to the connection identifier of the active call which is to be transferred.
<code>transferredTo</code>	[mandatory] A pointer to the destination address to which the call will be transferred.

`videoCall`

[optional] contains Call Type. This parameter used to convey to end points media type of the call .This parameter only applies to the `attV15SingleStepTransferCallConfEvent()` private data formatting function, which is available beginning with private data version 15.

The following media type are supported

- `VID_DISABLE` (Default) – media type as Audio
- `VID_INCOMING` – media type as audio irrespective of capabilities of end point. Will receive video of other party/call initiate party but party transfer call does not send video
- `VID_TWOWAY` – media type as Video. Both party will support as video.

Ack Parameters:

None for this service.

Ack Private Parameters:

`transferredCall`

[mandatory] Specifies the connection ID for the destination of the transferred call.

`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 only supported for private data version 9 and later.

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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 request failed for one of the following reasons:
 - 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 request failed for one of the following reasons:
 - 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 AE Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, etc.) for the same device.
 - One or more of the transfer participants is currently participating in another Single Step Transfer Call service request. The application should re-attempt the failed Single Step Transfer Call service request after 100-200 milliseconds.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `activeCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 15 and Later Syntax

```

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

/*
 * att15SingleStepTransferCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV15SingleStepTransferCall(
    ATTPrivateData_t      *privateData,
    ConnectionID_t        *activeCall,          /*
mandatory */
    DeviceID_t            *transferredTo,      /* mandatory */
    ATTVideoCall_t        videocall);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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;
} ATTEvent_t;

typedef enum ATTVideoCall_t {
    vidDisabled = 0,
    vidIncoming = 1,
    vidTwoway = 2,
} ATTVideoCall_t;

typedef char ATTUCID_t[64];

typedef struct ATTSingleStepTransferCallConfEvent_t {
    ConnectionID_t      transferredCall;

```

Single Step Transfer Call (Private Data Version 8 and later)

```
    ATTUCID_t          ucid;  
} ATTSingleStepTransferCallConfEvent_t;
```

Private Data Version 9-14 Syntax

```

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

/*
 * attV9SingleStepTransferCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV9SingleStepTransferCall(
    ATTPrivateData_t      *privateData,
    ConnectionID_t        *activeCall,          /*
mandatory */
    DeviceID_t            *transferredTo);      /*
mandatory */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTSingleStepTransferCallConfEvent_t {
    ConnectionID_t      transferredCall;
    ATTUCID_t           ucid;
} ATTSingleStepTransferCallConfEvent_t;

```

Private Data Version 8 Syntax

```

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

/*
 * attSingleStepTransferCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attSingleStepTransferCall(
    ATTPrivateData_t      *privateData,
    ConnectionID_t        *activeCall,          /*
mandatory */
    DeviceID_t            *transferredTo);      /*
mandatory */

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/*
 * ATTV8SingleStepTransferCallConfEvent - Private Data Service
Response
 */

typedef struct
{
    ATTEventType_t      eventType;

    ATTV8_SINGLE_STEP_TRANSFER_CALL_CONF /*
union
{
    ATTV8SingleStepTransferCallConfEvent_t
v8ssTransferCallConf;
} u;

} ATTEvent_t;

typedef struct ATTV8SingleStepTransferCallConfEvent_t {
    ConnectionID_t      transferredCall;
} ATTV8SingleStepTransferCallConfEvent_t;

```


Transfer Call Service

Summary

- Direction: Client to Switch
- Function: `cstaTransferCall()`
- Confirmation Event: `CSTATransferCallConfEvent`
- Private Data Confirmation Event: `ATTTransferCallConfEvent` (private data version 5 and later)
- Service Parameters: `heldCall`, `activeCall`
- Ack Parameters: `newCall`, `connList`
- Ack Private Parameters: `ucid`
- Nak Parameters: `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:

<code>heldCall</code>	[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 <code>activeCall</code> . The <code>deviceID</code> in <code>heldCall</code> must contain the station extension of the controlling device.
<code>activeCall</code>	[mandatory] Must be a valid connection identifier of an active or proceeding call at the controlling device to which the <code>heldCall</code> is to be transferred. The <code>deviceID</code> in <code>activeCall</code> must contain the station extension of the controlling device.

Ack Parameters:

<code>newCall</code>	[mandatory – partially supported] A connection identifier that specifies the resulting new call identifier for the transferred call.
<code>connList</code>	[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 <code>connectionIDs</code> and up to six <code>deviceIDs</code> 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 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). A group extension may contain alerting extensions. • The static <code>deviceID</code> 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:

<code>ucid</code>	[optional] Specifies the Universal Call ID (UCID) of <code>newCall</code> . 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 <code>ucid</code> contains the <code>ATT_NULL_UCID</code> (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CSTA_DEVICE_IDENTIFIER` (12) – An invalid device identifier or extension was 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) – The request failed for one of the following reasons:
 - Both calls are alerting.
 - Both calls are being service-observed.
 - An active call is in a vector-processing stage.
 - The Trunk-to-Trunk Transfer feature is not enabled on Avaya Communication Manager.
- `INVALID_OBJECT_STATE` (22) – The connections specified in the request are not in valid states for the operation to take place. For example, the transferring device 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 AE Services servers are issuing requests (for example, Hold Call, Retrieve Call, Clear Connection, Transfer Call, etc.) for the same device.
- `CONFERENCE_MEMBER_LIMIT_EXCEEDED` (38) – The request attempted to add a seventh party to an existing six-party conference call or thirteenth party to a twelve-party conference call (for ASAI link version 11 and above and 12 party conference enabled on CM).
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `activeCall` and `heldCall` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

- MISTYPED_ARGUMENT_REJECTION (74) – DYNAMIC_ID is specified in heldCall or activeCall.

Detailed Information:

- Analog Stations – The Transfer Call Service will only be allowed if one call is held and the second is active (talking). Calls on hard-held or that are alerting cannot be affected by the Transfer Call Service.
 - An analog station will support the 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 – The 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 cases, a transfer requested via the 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;
} CSTAEvent_t;

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

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

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

```

Private Data Version 5 and Later Syntax

```
/*
 * ATTransferCallConfEvent - Service Response Private Data
 * (private data version 5 and later)
 */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_TRANSFER_CALL_CONF */
    union
    {
        ATTransferCallConfEvent_t    transferCall;
    } u;
} ATTEvent_t;

typedef char ATTUCID_t[64];

typedef struct ATTransferCallConfEvent_t
{
    ATTUCID_t    ucid;
} ATTransferCallConfEvent_t;
```

Chapter 7: Set Feature Service Group

The Set Feature Service Group provides services that allow a client application to set switch-controlled features or values associated with a Communication Manager device. The following sections describe the Set Feature services supported by AE Services:

- [Set Advice of Charge Service \(Private Data Version 5 and Later\)](#) on page 435
- [Set Agent State Service](#) on page 439
- [Set Billing Rate Service \(Private Data Version 5 and Later\)](#) on page 452
- [Set Do Not Disturb Feature Service](#) on page 457
- [Set Forwarding Feature Service](#) on page 460
- [Set Message Waiting Indicator \(MWI\) Feature Service](#) on page 464

Set Advice of Charge Service (Private Data Version 5 and Later)

Summary

- Direction: Client to Switch
- Function: `cstaEscapeService()`
- Confirmation Event: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attSetAdviceOfCharge()`
- Service Parameters: `noData`
- Private Parameters: `flag`
- Ack Parameters: `noData`
- Ack Private Parameters: `noData`
- Nak Parameters: `universalFailure`

Functional Description

This service enables Communication Manager to support the collection of charging units over ISDN Primary Rate Interfaces. See [Detailed Information](#) for more information about this 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).

When 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 provide 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:

None for this service.

Private Parameters:

<code>flag</code>	[mandatory] Specify the flag for turning the feature on or off. A value of <code>TRUE</code> will turn the feature on and a value of <code>FALSE</code> 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 already turned off, subsequent requests to turn the feature off again will receive positive acknowledgements.

Ack Parameters:

None for this service.

Ack Private Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.
- `INVALID_FEATURE` (15) – The Set Advice of Charge Service is not supported by the switch.
 - `VALUE_OUT_OF_RANGE` (3) – The `flag` parameter value is invalid.

Detailed Information:

- The result of a successful Set Advice of Charge Service request applies to an ACS Stream. This means that any monitor 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) for Avaya Communication Manager.
- 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);

/* CSTAEscapeSvcConfEvent - 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 Syntax

```

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

/*
 * attSetAdviceOfCharge() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attAdviceOfCharge(
    ATTPrivateData *privateData,
    Boolean flag);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short 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 version 6 and later), `attSetAgentStateExt` (private data version 5 and later), `attSetAgentState` (private data version 2 and later)
- Service Parameters: `device`, `agentMode`, `agentID`, `agentGroup`, `agentPassword`
- Private Parameters: `workMode`, `reasonCode` (private data version 5 and later), `enablePending` (private data version 6 and later)
- Ack Parameters: `noData`
- Ack Private Parameters: `isPending`
- Nak Parameters: `universalFailure`

Functional Description:

This service allows a client to log an ACD agent into or out of an ACD Split and to specify a change of work mode for an ACD agent.

Service Parameters:

<code>device</code>	[mandatory] Specifies the agent extension. This must be a valid on-PBX station extension for an ACD agent.
<code>agentMode</code>	[mandatory – partially supported] Specifies whether to log an Agent into or out of an ACD split, or to change of work mode for an Agent logged into an ACD split: • <code>AM_LOG_IN</code> – 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 <code>workMode</code> private parameter. If the <code>workMode</code> private parameter is not supplied, the initial work mode for the ACD agent will be set to "Auxiliary-Work Mode". • <code>AM_LOG_OUT</code> – Log an Agent out of a specific ACD split. The Agent will be unable to accept additional calls for the ACD split. • <code>AM_NOT_READY</code> – Change the work mode for an Agent logged into an ACD split to "Not Ready" (equivalent to the Communication Manager "Auxiliary-Work Mode"), indicating that the Agent is occupied with some task other than serving a call. • <code>AM_READY</code> – Change the work mode for Agent logged into an ACD split to "Ready". An Agent in the Ready state is ready to accept calls or is currently busy with an ACD call. The <code>workMode</code> private parameter may be used to set the ACD agent work mode to "Auto-In-Work Mode" or "Manual-In-Work Mode". If the <code>workMode</code> private parameter is not supplied, the ACD agent work mode will be set to "Auto-In-Work Mode". • <code>AM_WORK_NOT_READY</code> – Change the work mode for an Agent logged into an ACD split to "Work Not Ready" (equivalent to Communication Manager "After-Call-Work Mode"). An 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. • <code>AM_WORK_READY</code> – A change to "Work Ready" is not currently supported for Communication Manager.
<code>agentID</code>	[optional] Specifies the Agent login identifier for the ACD agent. This parameter is mandatory when the <code>agentMode</code> parameter is <code>AM_LOG_IN</code> ; otherwise it is ignored. An <code>agentID</code> containing a Logical Agent's login Identifier can be used to log in a Logical Agent (Expert Agent Selection [EAS] environment) when paired with the <code>agentPassword</code> .
<code>agentGroup</code>	[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 <code>agentGroup</code> parameter is ignored and in that case is optional.

`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 are `WM_AUTO_IN` (Auto-In-Work Mode, theDefault), and `WM_MANUAL_IN` (Manual-In-Work Mode).
- `AM_WORK_NOT_READY` – The `workMode` is ignored.
- `AM_WORK_READY` – The `workMode` is ignored.

Refer to [Table 13](#) for valid combinations of the `agentMode` service parameter and the `workMode`, `reasonCode`, and `enablePending` private parameters.

`reasonCode` [optional] Specifies the reason for a work mode change to `WM_AUX_WORK` or the logged-out (`AM_LOG_OUT`) state.

Beginning with 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.

Private data versions 5 and 6 support 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 versions 4 and earlier do not support reason codes.

Refer to [Table 13](#) for valid combinations of the `agentMode` service parameter and the `workMode`, `reasonCode`, and `enablePending` private parameters.

`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_READY`, `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.
 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 fail.

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 versions 6 and later.
 Refer to [Table 13](#) for valid combinations of the `agentMode` service parameter and the `workMode`, `reasonCode`, and `enablePending` private parameters.

Ack Parameters:

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 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `GENERIC_UNSPECIFIED (0)`

The request has failed for one of the following reasons:

- The request attempted to log out an ACD agent who is already logged out.
- The request attempted to log an ACD agent into a split of which they are not a member.
- The request attempted to log in an ACD agent with an incorrect password.
- The request attempted to log in an ACD agent at a station where the Auto Answer feature is enabled, but the station is not off-hook idle.

- `GENERIC_OPERATION (1)` – The request attempted to log in an ACD agent that is already logged in.

- `VALUE_OUT_OF_RANGE (3)`

The request has failed for one of the following reasons:

- The `workMode` private parameter is not valid for the `agentMode` (see [Table 13](#)).
- The reason code is outside of the acceptable range (1- 9 or 1-99). (CS0/100).

- `OBJECT_NOT_KNOWN (4)` – The service request 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 switch version.

- `INVALID_OBJECT_TYPE (18) (CS3/80)` – A reason code was specified, but the specified `workMode` was not `WM_AUX_WORK`, or `agentMode` was 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, or the maximum number of agents is already logged in.

- `GENERIC_SYSTEM_RESOURCE_AVAILABILITY (31)` – The request cannot be completed due to lack of available switch resources.
- `RESOURCE_BUSY (33)` – The service attempted to change the state of an ACD agent that is currently on a call.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41)` – The device identifier specified in `device` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

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 (in an Expert Agent Selection [EAS] environment) 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	NA – ignored	1-99	NA
AM_NOT_READY	NA – ignored	1-99	TRUE/FALSE
AM_READY	WM_AUTO_IN (default)	NA	TRUE/FALSE
	WM_MANUAL_IN		
AM_WORK_NOT_READY	NA – ignored	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.
- Subsequent pending work mode requests supersede any 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];

/* CSTASetAgentStateConfEvent - Service Response */

typedef struct
{
    ACSHandle_t    acsHandle;
    EventClass_t   eventClass;           /* CSTACONFIRMATION */
    EventType_t    eventType;           /*
CSTA_SET_AGENT_STATE_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            InvokeID_t invokeID;

```

```
        union
        {
            CSTASetAgentStateConfEvent_t  setAgentState;
        } u;
    } cstaConfirmation;
} event;
} CSTAEvent_t;

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

Private Data Version 6 and Later Syntax

```

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

/*
 * attV6SetAgentState() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV6SetAgentState(
    ATTPrivateData_t      *privateData,
    ATTWorkMode_t          workMode,
    long                   reasonCode,          /*
1-9 for private data
                                           * version 6, 1-99 for
                                           * version 7 and
                                           * later */
    Boolean                enablePending);      /* TRUE =
enabled */

typedef struct ATTPrivateData_t {
    char                   vendor[32];
    unsigned short         length;
    char                   data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTWorkMode_t
{
    WM_NONE = -1,           /* not specified */
    WM_AUX_WORK = 1,        /* Auxiliary Work Mode*/
    WM_AFTCAL_WK = 2,       /* After Call Work Mode*/
    WM_AUTO_IN = 3,         /* Auto In Mode*/
    WM_MANUAL_IN = 4        /* Manual In Mode*/
} ATTWorkMode_t;

/* ATTSetAgentStateConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t        eventType; /* ATT_SET_AGENT_STATE_CONF */
    union
    {
        ATTSetAgentStateConfEvent_t setAgentState;
    }u;
} ATTEvent_t;

```

```
typedef struct ATTSetAgentStateConfEvent_t {  
    unsigned char    isPending; /* TRUE if request is pending */  
} ATTSetAgentStateConfEvent_t;
```

Private Data Version 5 Syntax

```

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

/* attSetAgentStateExt() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attSetAgentStateExt(
    ATTPrivateData_t *privateData,
    ATTWorkMode_t workMode,
    long reasonCode); /* single
digit 1-9 */

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTWorkMode_t
{
    WM_NONE = -1, /* not specified */
    WM_AUX_WORK = 1, /* Auxiliary Work Mode*/
    WM_AFTCAL_WK = 2, /* After Call Work Mode*/
    WM_AUTO_IN = 3, /* Auto In Mode*/
    WM_MANUAL_IN = 4 /* Manual In Mode*/
} ATTWorkMode_t;

```


Private Data Versions 2-4 Syntax

```

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

/* attSetAgentState() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attSetAgentState(
    ATTPrivateData_t *privateData,
    ATTWorkMode_t workMode);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef enum ATTWorkMode_t
{
    WM_NONE = -1, /* not specified */
    WM_AUX_WORK = 1, /* Auxiliary Work Mode*/
    WM_AFTCAL_WK = 2, /* After Call Work Mode*/
    WM_AUTO_IN = 3, /* Auto In Mode*/
    WM_MANUAL_IN = 4 /* Manual In Mode*/
} ATTWorkMode_t;

```

Set Billing Rate Service (Private Data Version 5 and Later)

Summary

- Direction: Client to Switch
- Function: `cstaEscapeService()`
- Confirmation Event: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attSetBillRate()`
- Service Parameters: `noData`
- Private Parameters: `call, billType, billRate`
- Ack Parameters: `noData`
- Nak Parameters: `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:

None for this service.

Private Parameters:

<code>call</code>	[mandatory] Specifies the call to which the billing rate is to be applied. This is a connection identifier, but only the <code>callID</code> is used. The <code>deviceID</code> of the call is ignored.
<code>billType</code>	[mandatory] Specifies the rate treatment for the call and can be one of the following: • <code>BT_NEW_RATE</code> • <code>BT_FLAT_RATE</code> (time independent) • <code>BT_PREMIUM_CHARGE</code> (i.e., a flat charge in addition to the existing rate) • <code>BT_PREMIUM_CREDIT</code> (i.e., a flat negative charge in addition to the existing rate) • <code>BT_FREE_CALL</code>
<code>billRate</code>	[mandatory] Specifies the rate according to the treatment indicated by <code>billType</code> . If <code>FREE_CALL</code> is specified, <code>billRate</code> 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:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CSTA_CONNECTION_IDENTIFIER` (13) – An invalid connection identifier has been specified in call.
- `VALUE_OUT_OF_RANGE` (3) (CS0/96) – An invalid value has been specified in the request.
- `INVALID_OBJECT_STATE` (22) (CS0/98) – The request was attempted before the call was 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 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);

/* CSTAEscapeSvcConfEvent - 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 Syntax

```

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

/*
 * attSetBillRate() - Service Request Private Data Formatting Function
 */

RetCode_t attSetBillRate(
    ATTPrivateData      *privateData,
    ConnectionID_t      *call,
    ATTBillType_t        billType,
    float                billRate);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short       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 Parameters: `universalFailure`

Functional Description:

This service turns on or off the Communication Manager Send All Calls (SAC) feature for a user station.

Service Parameters:

<code>device</code>	[mandatory] Must be a valid on-PBX station extension that supports the SAC feature.
<code>doNotDisturb</code>	[mandatory] Specifies either "On" (<code>TRUE</code>) or "Off" (<code>FALSE</code>).

Ack Parameters:

None for this service.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The <code>error</code> parameter in this event may contain the following error values, or one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997. • <code>INVALID_CSTA_DEVICE_IDENTIFIER</code> (12) – An invalid device identifier has been specified in <code>device</code>. • <code>GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY</code> (41) – The request failed for one of the following reasons: – The user has not subscribed to the Send All Calls (SAC) feature. – The device identifier specified in <code>device</code> corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

- DCS – The SAC feature may not be requested by this service for an off-PBX DCS extension.
- Do Not Disturb – Despite its name the `cstaSetDoNotDisturb()` service controls the Send All Calls (SAC) feature; not the Do Not Disturb feature.
- Logical Agents – The SAC feature 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`). The SAC feature 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, 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 Communication Manager 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 Avaya Aura Communication Manager 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, 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
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTASetDndConfEvent_t  setDnd;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASetDndConfEvent_t {
    Nulltype        null;
} CSTASetDndConfEvent_t;

```

Set Forwarding Feature Service

Summary

- Direction: Client to Switch
- Function: `cstaSetForwarding()`
- Confirmation Event: `CSTASetFwdConfEvent`
- Service Parameters: `device`, `forwardingType`, `forwardingOn`, `forwardingDestination`
- Ack Parameters: `noData`
- Nak Parameters: `universalFailure`

Functional Description:

The Set Forwarding Service sets the Communication Manager Call Forwarding feature on or off for a user station. For Communication Manager, this service only supports the Immediate type of forwarding.

Service Parameters:

<code>device</code>	[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.
<code>forwardingType</code>	[mandatory – partial] Specifies the type of forwarding to set or clear. Only <code>FWD_IMMEDIATE</code> is supported. Any other types will be denied.
<code>forwardingOn</code>	[mandatory] Specifies "On" (<code>TRUE</code>) or "Off" (<code>FALSE</code>).
<code>forwarding-Destination</code>	[mandatory] Specifies the station extension to which the calls are to be forwarded. It is mandatory if <code>forwardingOn</code> is set to <code>TRUE</code> . It is ignored by Communication Manager if the <code>forwardingOn</code> is set to <code>FALSE</code> .

Ack Parameters:

None for this service.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The <code>error</code> parameter in this event may contain the following error values, or one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997. • <code>INVALID_CSTA_DEVICE_IDENTIFIER</code> (12) – An invalid device identifier has been specified for <code>device</code> or <code>forwardingDestination</code>. • <code>GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY</code> (41) – The request failed for one of the following reasons: – The user is not subscribed to the Call Forwarding feature. – The device identifier specified in <code>device</code> corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

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 forwarding destination, the request will be denied (`INVALID_CSTA_DEVICE_IDENTIFIER`). Call Forwarding may be requested on behalf of a logical agent’s station extension.
- Communication Manager 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, FALSE = Off */
    DeviceID_t     *forwardingDestination,
    PrivateData_t  *privateData);

typedef char    DeviceID_t[64];

typedef enum ForwardingType_t {
    FWD_IMMEDIATE = 0, /* The only supported type */
    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
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            InvokeID_t invokeID;
            union
            {

```

```
                CSTASetFwdConfEvent_t    setFwd;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASetFwdConfEvent_t {
    Nulltype    null;
} CSTASetFwdConfEvent_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 Parameters: `universalFailure`

Functional Description:

This service sets the Message Waiting Indicator (MWI) on or off for a user station.

Service Parameters:

<code>device</code>	[mandatory] Must be a valid on-PBX station extension that supports the MWI feature.
<code>messages</code>	[mandatory] Specifies either "On" (<code>TRUE</code>) or "Off" (<code>FALSE</code>).

Ack Parameters:

None for this service.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The <code>error</code> parameter in this event may contain the following error value, or one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997. • <code>INVALID_CSTA_DEVICE_IDENTIFIER</code> (12) – An invalid device identifier has been specified in <code>device</code>. • <code>GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY</code> (41) – The device identifier specified in <code>device</code> corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

- Adjunct Messages – When a client application has turned on a station's MWI and the station user retrieves the 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 TSAPI link between the switch and the AE Services server comes up from a cold start. An application can query the MWI status using 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, 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
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTASetMwiConfEvent_t  setMwi;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASetMwiConfEvent_t {
    Nulltype        null;
} CSTASetMwiConfEvent_t;

```


Chapter 8: Query Service Group

The Query Service Group provides services that allow a client application to query the switch to provide the state of device features and to retrieve static attributes of a device. The following sections describe the Query services supported by AE Services:

- [Query ACD Split Service](#) on page 469
- [Query Agent Login Service](#) on page 473
-
-
-
- [Query Agent State Service](#) on page 483
- [Query Call Classifier Service](#) on page 493
- [Query Device Info](#) on page 497
- [Query Device Name Service](#) on page 504
- [Query Do Not Disturb Service](#) on page 511
- [Query Endpoint Registration Info Service \(Private Data Version 11 and later\)](#) on page 514
- [Query Forwarding Service](#) on page 522
- [Query Message Waiting Indicator Service](#) on page 526
- [Query Station Status Service](#) on page 531
- [Query Time of Day Service](#) on page 536
- [Query Trunk Group Service](#) on page 540
- [Query Universal Call ID Service \(Private\)](#) on page 544

Query ACD Split Service

Summary

- Direction: Client to Switch
- Function: `cstaEscapeService()`
- Confirmation Event: `CSTAEscapeSvcConfEvent`
- 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 Parameters: `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:

None for this service.

Private Parameters:

<code>device</code>	[mandatory] Must be a valid ACD split extension.

Ack Parameters:

None for this service.

Ack Private Parameters:

<code>availableAgents</code>	[mandatory] Specifies the number of ACD agents available to receive calls through the specified split.
<code>callsInQueue</code>	[mandatory] Specifies the number of calls in queue (not including direct-agent calls).
<code>agentsLoggedIn</code>	[mandatory] Specifies the number of ACD agents logged in.

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CSTA_DEVICE_IDENTIFIER` (12) – An invalid device identifier has been specified in device.

Detailed Information:

None for this service.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 Syntax

```

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

/*
 * attQueryAcdSplit() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryAcdSplit(
    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_t eventType; /* ATT_QUERY_ACD_SPLIT_CONF */
    union
    {
        ATTQueryAcdSplitConfEvent_t queryAcdSplit;
    }u;
} ATTEvent_t;

typedef struct ATTQueryAcdSplitConfEvent_t {
    short availableAgents; /* number of agents available
                           * to
receive calls */
    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: `CSTAEscapeSvcConfEvent`
- Event Report: `CSTAPrivateEvent`
- Private Data Function: `attQueryAgentLogin()`
- Private Data Confirmation Event: `ATTQueryAgentLoginConfEvent`
- Private Data Event Report: `ATTV14QueryAgentLoginResp`,
`ATTVQueryAgentLoginResp`
- Service Parameters: `noData`
- Private Parameters: `device`
- Ack Parameters: `noData`
- Ack Private Parameters: `privEventCrossRefID`
- Private Event Parameters: `privEventCrossRefID`, `list`, `skillInfolist`
- Nak Parameters: `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:

None for this service.

Private Parameters:

`device` [mandatory] Must be a valid ACD split extension.

Ack Parameters:

None for this service.

Ack Private Parameters:

`privEventCrossRefID` Contains a unique handle to identify subsequent `CSTAPrivateEvents` associated 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 indicates that there are no additional `CSTAPrivateEvents` for the query.

skillInfolist

[optional] skillInfolist is a structure which has AgentSkillInfo and count.

count – stores size of ATTAgentSkillInfo_t array.

AgentSkillInfo – pointer to array of ATTAgentSkillInfo_t

Within ATTAgentSkillInfo_t:

- device – is a pointer to a null-terminated string that specifies the extension of the physical.
- agentID – is a pointer to a null-terminated string that specifies the extension of the logical agent ID.
- skillLevel is a structure of the type ATTSkillLevel_t

Within skillLevel:

- type – indicates the skill level type. This field is set to SKILL_LEVEL_TYPE_REGULAR if the level provided is a normal skill level. This field is set to SKILL_LEVEL_TYPE_RESERVED if the level provided is a reserved skill level. If no skill level information is provided, this field is set to SKILL_LEVEL_TYPE_NOT_PROVIDED.
- level – indicates the skill level based on the type. If type is set to SKILL_LEVEL_TYPE_NOT_PROVIDED, this field is undefined and should not be used.
- percentAllocation – This parameter indicates the agent's percent allocation, if configured on CM, and can range from 1 to 100. If not configured on CM, or if not received from CM, it will default to 0.

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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors - universalFailure](#) on page 997.

- INVALID_CSTA_DEVICE_IDENTIFIER (12) – An invalid device identifier has been specified in device.

Detailed Information:

- A single Query Agent Login service request may result in multiple `CSTAPrivateEvents` returned to the client after the return of the confirmation event. All messages are contained in the 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_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t   *privateData);

/* CSTAEscapeSvcConfEvent - 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;

/* CSTAPrivateEvent - Private event for reporting data */

typedef struct
{
    ACSHandle_t    acsHandle;
    EventClass_t    eventClass;           /* CSTAEVENTREPORT */
    EventType_t    eventType;           /* CSTA_PRIVATE */
} ACSEventHeader_t;

```

```

typedef struct
{
    ACSEEventHeader_t  eventHeader;
    union
    {
        struct
        {
            union
            {
                CSTAPrivateEvent_t  privateEvent;
            } u;
        } cstaEventReport;
    } event;
} CSTAEvent_t;

typedef struct CSTAPrivateEvent_t {
    Nulltype      null;
} CSTAPrivateEvent_t;

```

Private Data Version 15 and Later

```

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

/*
 * attQueryAgentLogin() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryAgentLogin(
    ATTPrivateData_t *privateData,
    DeviceID_t *device);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTQueryAgentLoginConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t eventType; /* ATT_QUERY_AGENT_LOGIN_CONF */
    union
    {
        ATTQueryAgentLoginConfEvent_t queryAgentLogin;
    } u;
} ATTEvent_t;

typedef long ATTPrivEventCrossRefID_t;

typedef struct ATTQueryAgentLoginConfEvent_t {
    ATTPrivEventCrossRefID_t privEventCrossRefID;
} ATTQueryAgentLoginConfEvent_t;

/* ATTQueryAgentLoginResp - CSTA Private Event Private Data */

typedef struct
{
    ATTEventType_t eventType; /* ATT_QUERY_AGENT_LOGIN_RESP */
    union
    {
        ATTQueryAgentLoginResp_t queryAgentLoginResp;
    } u;
} ATTEvent_t;

```

```

typedef struct ATTQueryAgentLoginResp_t {
    ATTPrivEventCrossRefID_t    privEventCrossRefID;
    struct {
        unsigned short          count;                /* number of
extensions in

        * device[] */
        DeviceID_t              device[10]; /* up to 10 extensions
*/
    } list;
    struct {
        unsigned short count;    /*Number of AgentSkillInfo in
AgentSkillInfo[] */
        ATTAgentSkillInfo_t AgentSkillInfo[10];    /*up to 10
AgentSkillInfo*
    } skillInfolist;
} ATTQueryAgentLoginResp_t;

typedef struct ATTAgentSkillInfo_t {
    DeviceID_t    device;
    AgentID_t     agentID;
    ATTSkillLevel_t skillLevel;
    unsigned char percentAllocation;
} ATTAgentSkillInfo_t;

typedef enum ATTSkillLevelType_t {
    skLvlTypeNotProvided = 0,
    skLvlTypeRegular = 1,
    skLvlTypeReserved = 2
} ATTSkillLevelType_t;

typedef struct ATTSkillLevel_t {
    ATTSkillLevelType_t type;
    long                level;
} ATTSkillLevel_t;

```

Private Data Version 14

```

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

/*
 * attQueryAgentLogin() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryAgentLogin(
    ATTPrivateData_t *privateData,
    DeviceID_t *device);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTQueryAgentLoginConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t eventType; /* ATT_QUERY_AGENT_LOGIN_CONF */
    union
    {
        ATTQueryAgentLoginConfEvent_t queryAgentLogin;
    } u;
} ATTEvent_t;

typedef long ATTPrivEventCrossRefID_t;

typedef struct ATTQueryAgentLoginConfEvent_t {
    ATTPrivEventCrossRefID_t privEventCrossRefID;
} ATTQueryAgentLoginConfEvent_t;

/* ATTQueryAgentLoginResp - CSTA Private Event Private Data */

typedef struct
{
    ATTEventType_t eventType; /* ATT_QUERY_AGENT_LOGIN_RESP */
    union
    {
        ATTQueryAgentLoginResp_t queryAgentLoginResp;
    } u;
} ATTEvent_t;

typedef struct ATTV14QueryAgentLoginResp_t {

```

```

ATTPrivEventCrossRefID_t    privEventCrossRefID;
struct {
    unsigned short          count;          /* number of
extensions in
    * device[] */
    DeviceID_t              device[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 versions 6 and later), `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` (private data version 5 and later), `pendingWorkMode`, `pendingReasonCode` (private data version 6 and later)
- Nak Parameters: `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 Avaya Communication Manager. 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:

<code>device</code>	[mandatory] Must be a valid agent extension or a logical agent ID.

Private Parameters:

<code>split</code>	[optional] If specified, it must be a valid ACD split extension. This parameter is optional in an Expert Agent Selection (EAS) environment, but it is mandatory in a non-EAS environment.

Ack Parameters:

<code>agentState</code>	[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.
- Communication Manager does not support the `AG_WORK_READY` state.

Ack Private Parameters:

<code>workMode</code>	[optional] This parameter provides the agent work mode as defined by Avaya Communication Manager. Valid values include: • <code>WM_AUTO_IN</code> – Indicates that the agent work mode is Auto-In. The agent is allowed to receive a new call immediately after disconnecting from the previous call. The <code>talkState</code> parameter indicates whether the agent is busy or idle. • <code>WM_MANUAL_IN</code> – Indicates that the agent work mode is Manual-In. The agent is automatically changed to the <code>WM_AFTCAL_WK</code> state immediately after disconnecting from the previous call. • <code>WM_AFTCAL_WK</code> – Indicates that the agent work mode is After Call Work. (A Query Agent State service request for an agent in this work mode returns an <code>agentState</code> parameter value of <code>AG_WORK_NOT_READY</code>.) • <code>WM_AUX_WORK</code> – Indicates that the agent work mode is Auxiliary Work. (A Query Agent State service request on an agent in this work mode returns an <code>agentState</code> parameter value of <code>AG_NOT_READY</code>.)
<code>talkState</code>	[optional] The <code>talkState</code> parameter provides the actual readiness of the agent. Valid values are: • <code>TS_ON_CALL</code> – Indicates that the agent is occupied with serving a call • <code>TS_IDLE</code> – Indicates that the agent is ready to accept calls.
<code>reasonCode</code>	[optional] Specifies the reason for change of work mode to <code>WM_AUX_WORK</code> or the logged-out (<code>AM_LOG_OUT</code>) state. Beginning with 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. Private data versions 5 and 6 support 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 versions 4 and earlier do not support reason codes.
<code>pendingWorkMode</code>	[optional] Specifies the work mode which will take effect when the agent gets off the call. If no work mode is pending then <code>pendingWorkMode</code> will be set to <code>WM_NONE</code> (-1).
<code>pendingReasonCode</code>	[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 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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors - - universalFailure](#) on page 997.
- `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 when the `agentState` has the value `AG_NULL`, all confirmation events include private parameters for the 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_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *device,
    PrivateData_t  *privateData);

/* CSTAQueryAgentStateConfEvent - Service Response */

typedef struct
{
    ACSHandle_t    acsHandle;
    EventClass_t   eventClass;    /* CSTACONFIRMATION */
    EventType_t    eventType;     /* CSTA_QUERY_AGENT_STATE_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAQueryAgentStateConfEvent_t
                queryAgentState;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAQueryAgentStateConfEvent_t {
    AgentState_t    agentState;
} 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 supported */
}

```

```
} AgentState_t;
```

Private Data Version 6 and Later Syntax

```

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

/*
 * attQueryAgentState() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryAgentState(
    ATTPrivateData_t *privateData,
    DeviceID_t *split);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTQueryAgentStateConfEvent - Service Response Private Data */

typedef struct ATTEvent_t
{
    ATTEventType_t eventType; /* ATT_QUERY_AGENT_STATE_CONF */
    union
    {
        ATTQueryAgentStateConfEvent_t queryAgentState;
    } u;
} ATTEvent_t;

typedef enum ATTWorkMode_t {
    WM_NONE = -1, /* No work mode is pending */
    WM_AUX_WORK = 1, /* Auxiliary Work Mode*/
    WM_AFTCAL_WK = 2, /* After Call Work Mode*/
    WM_AUTO_IN = 3, /* Auto In Mode*/
    WM_MANUAL_IN = 4 /* Manual In Mode*/
} ATTWorkMode_t;

typedef enum ATTTalkState_t {
    TS_ON_CALL = 0,
    TS_IDLE = 1
} ATTTalkState_t;

typedef struct ATTQueryAgentStateConfEvent_t {
    ATTWorkMode_t workMode;
    ATTTalkState_t talkState;
    long reasonCode;
    ATTWorkMode_t pendingWorkMode;
    long pendingReasonCode;
} ATTQueryAgentStateConfEvent_t;

```


Private Data Version 5 Syntax

```

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

/*
 * attQueryAgentState() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryAgentState(
    ATTPrivateData_t *privateData,
    DeviceID_t *split);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTV5QueryAgentStateConfEvent - Service Response Private Data */

typedef struct ATTEvent_t
{
    ATTEventType_t eventType; /* ATTV5_QUERY_AGENT_STATE_CONF */
    union
    {
        ATTV5QueryAgentStateConfEvent_t v5queryAgentState;
    } u;
} ATTEvent_t;

typedef enum ATTWorkMode_t {
    WM_NONE = -1, /* No work mode is pending */
    WM_AUX_WORK = 1, /* Auxiliary Work Mode*/
    WM_AFTCAL_WK = 2, /* After Call Work Mode*/
    WM_AUTO_IN = 3, /* Auto In Mode*/
    WM_MANUAL_IN = 4 /* Manual In Mode*/
} ATTWorkMode_t;

typedef enum ATTTalkState_t {
    TS_ON_CALL = 0,
    TS_IDLE = 1
} ATTTalkState_t;

typedef struct ATTV5QueryAgentStateConfEvent_t {
    ATTWorkMode_t workMode;
    ATTTalkState_t talkState;
    long reasonCode;
} ATTV5QueryAgentStateConfEvent_t;

```

Private Data Versions 2-4 Syntax

```

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

/*
 * attQueryAgentState() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryAgentState(
    ATTPrivateData_t      *privateData,
    DeviceID_t            *split);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTV4QueryAgentStateConfEvent - Service Response Private Data */

typedef struct ATTEvent_t
{
    ATTEventType_t      eventType; /* ATTV4_QUERY_AGENT_STATE_CONF */
    union
    {
        ATTV4QueryAgentStateConfEvent_t      v4queryAgentState;
    } u;
} ATTEvent_t;

typedef enum ATTWorkMode_t {
    WM_NONE = -1, /* No work mode is pending */
    WM_AUX_WORK = 1, /* Auxiliary Work Mode*/
    WM_AFTCAL_WK = 2, /* After Call Work Mode*/
    WM_AUTO_IN = 3, /* Auto In Mode*/
    WM_MANUAL_IN = 4 /* Manual In Mode*/
} ATTWorkMode_t;

typedef enum ATTTalkState_t {
    TS_ON_CALL = 0,
    TS_IDLE = 1
} ATTTalkState_t;

typedef struct ATTV4QueryAgentStateConfEvent_t {
    ATTWorkMode_t      workMode;
    ATTTalkState_t      talkState;
} ATTV4QueryAgentStateConfEvent_t;

```

Query Call Classifier Service

Summary

- Direction: Client to Switch
- Function: `cstaEscapeService()`
- Confirmation Event: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attQueryCallClassifier()`
- Private Data Confirmation Event: `ATTQueryCallClassifierConfEvent`
- Service Parameters: `noData`
- Private Parameters: `noData`
- Ack Parameters: `noData`
- Ack Private Parameters: `numAvailPorts, numInUsePorts`
- Nak Parameters: `universalFailure`

Functional Description:

This service provides the number of "idle" and "in-use" call classifier (e.g., TN744) ports. The "in use" number is a snapshot of the call classifier port usage.

Service Parameters:

None for this service.

Private Parameters:

None for this service.

Ack Parameters:

None for this service.

Ack Private Parameters:

<code>numAvailPorts</code>	[mandatory] The number of available ports.
<code>numInUsePorts</code>	[mandatory] The number of "in use" ports.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code> . The <code>error</code> parameter in this event may one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997.

Detailed Information:

None for this service.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 Syntax

```

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

/*
 * attQueryCallClassifier() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryCallClassifier(
    ATTPrivateData_t *privateData);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTQueryCallClassifierConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t eventType; /* ATT_QUERY_CALL_CLASSIFIER_CONF
*/
    union
    {
        ATTQueryCallClassifierConfEvent_t queryCallClassifier;
    } u;
} ATTEvent_t;

typedef struct ATTQueryCallClassifierConfEvent_t {
    short numAvailPorts; /* number of available ports
*/
    short numInUsePorts; /* number of ports in use */
} ATTQueryCallClassifierConfEvent_t;

```

Query Device Info

Summary

- Direction: Client to Switch
- Function: `cstaQueryDeviceInfo()`
- Confirmation Event: `CSTAQueryDeviceInfoConfEvent`
- Private Data Confirmation Event: `ATTQueryDeviceInfoConfEvent` (private data version 5 and later), `ATTV4QueryDeviceInfoConfEvent` (private data versions 2-4)
- Service Parameters: `device`
- Ack Parameters: `device`, `deviceType`, `deviceClass`
- Ack Private Parameters: `extensionClass`, `associatedDevice` (private data version 5 and later), `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 extension class is provided in the CSTA private data.

Service Parameters:

<code>device</code>	[mandatory] Must be a valid on-PBX station extension.

Ack Parameters:

<code>device</code>	[optional – supported] Normally this is the same ID specified in the device parameter in the request. See <code>associatedDevice</code> and <code>associatedClass</code> below.
<code>deviceType</code>	[mandatory] The device type (mapped from the Communication Manager extension class).
<code>deviceClass</code>	[mandatory] The device class (mapped from the Communication Manager extension class).

Ack Private Parameters:

<code>extensionClass</code>	[mandatory] The Communication Manager Extension Class for the device.
<code>associatedDevice</code>	[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. Similarly, 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, an empty string is returned. This parameter is supported by private data version 5 and later only.
<code>associatedClass</code>	[optional] The Communication Manager Extension Class for the <code>associatedDevice</code> . Its value is <code>EC_LOGICAL_AGENT</code> if the <code>associatedDevice</code> is a device ID of a logical agent; otherwise its value is <code>EC_OTHER</code> . This parameter is supported by private data version 5 and later only.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The <code>error</code> parameter in this event may contain the following error value, or one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997. • <code>INVALID_CSTA_DEVICE_IDENTIFIER</code> (12) – An invalid device identifier has been specified in <code>device</code>. • <code>GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY</code> (41) – The device identifier specified in <code>device</code> corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

The `deviceType` and `deviceClass` parameters are mapped from the Communication Manager extension class as follows:

Communication Manager Extension Class	CSTA Device Class	CSTA Device Type
VDN	Voice ³	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

³ 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_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *device,
    PrivateData_t  *privateData);

/* CSTAQueryDeviceInfoConfEvent - Service Response */

typedef struct
{
    ACSHandle_t    acsHandle;
    EventClass_t   eventClass;           /* CSTACONFIRMATION */
    EventType_t    eventType;           /*
CSTA_QUERY_DEVICE_INFO_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAQueryDeviceInfoConfEvent_t
queryDeviceInfo;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAQueryDeviceInfoConfEvent_t {
    DeviceID_t    device;
    DeviceType_t   deviceType;
    DeviceClass_t  deviceClass;
} CSTAQueryDeviceInfoConfEvent_t;

/* Device Types */

typedef enum DeviceType_t {
    DT_STATION = 0,

```

```

        DT_LINE = 1,                                /* not expected for
Avaya CM */
        DT_BUTTON = 2,                              /* not expected for
Avaya CM */
        DT_ACD = 3,
        DT_TRUNK = 4,                                /* not expected for
Avaya CM */
        DT_OPERATOR = 5,                            /* not expected for Avaya CM
*/
        DT_STATION_GROUP = 16,                      /* not expected for Avaya CM */
        DT_LINE_GROUP = 17,                         /* not expected for Avaya CM
*/
        DT_BUTTON_GROUP = 18,                      /* not expected for Avaya CM */
        DT_ACD_GROUP = 19,
        DT_TRUNK_GROUP = 20,                        /* not expected for Avaya CM
*/
        DT_OPERATOR_GROUP = 21,                    /* not expected for Avaya CM */
        DT_OTHER = 255
} DeviceType_t;

typedef unsigned char    DeviceClass_t;

/* Device Classes */

#define DC_VOICE          0x80
#define DC_DATA           0x40
#define DC_IMAGE          0x20    /* not expected for Avaya CM */
#define DC_OTHER          0x10

```

Private Data Version 5 and Later Syntax

```

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

/* ATTQueryDeviceInfoConfEvent - Service Response Private Data */
typedef struct ATTEvent_t
{
    ATTEventType_t    eventType; /* ATT_QUERY_DEVICE_INFO_CONF */
    union
    {
        ATTQueryDeviceInfoConfEvent_t queryDeviceInfo;
    } u;
} ATTEvent_t;

typedef struct ATTQueryDeviceInfoConfEvent_t {
    ATTExtensionClass_t    extensionClass;
    ATTExtensionClass_t    associatedClass;
    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_t    eventType; /* ATTV4_QUERY_DEVICE_INFO_CONF */
    union
    {
        ATTV4QueryDeviceInfoConfEvent_t    v4queryDeviceInfo;
    } u;
} ATTEvent_t;

typedef struct ATTV4QueryDeviceInfoConfEvent_t {
    ATTExtensionClass_t    extensionClass;
} 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 and later), `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 to determine the name associated with a device. The name is retrieved from the Communication Manager Integrated Directory Database.

This service allows an application to identify the names administered in Communication Manager for device extension numbers without maintaining its own database.

Service Parameters:

None for this service.

Private Parameters:

<code>device</code>	[mandatory] Must be a valid device extension.

Ack Parameters:

None for this service.

Ack Private Parameters:

`deviceType` [mandatory] Specifies the device type of the device:

- `ATT_DT_UNKNOWN` – unknown
- `ATT_DT_ACD_SPLIT` – ACD Split (Hunt Group)
- `ATT_DT_ANNOUNCEMENT` – announcement
- `ATT_DT_DATA` – data extension
- `ATT_DT_LOGICAL_AGENT` – logical agent
- `ATT_DT_STATION` – station extension
- `ATT_DT_TRUNK_ACCESS_CODE` – Trunk Access Code
- `ATT_DT_VDN` – VDN
- `ATT_DT_OTHER` – The device type is something other than any of the device types listed above.

If no name is administered in Communication Manager for the device, the `attQueryDeviceName()` service sets the `deviceType` to `ATT_DT_UNKNOWN`.

`device` [mandatory] Specifies the extension number of the device.

**NOTE:**

If no name is administered in Communication Manager for the device, the `attQueryDeviceName()` service sets the device parameter to the empty string (`""`).

name

[mandatory] Specifies the associated name of the device. This is a string of 1-15 ASCII characters for private data versions 2-4. 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()` service sets the name parameter to the empty string ("").

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 the 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 their 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()` service sets the `uname` parameter to the empty string ("").

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `VALUE_OUT_OF_RANGE (3) (CS0/100)` – An invalid parameter value has been specified.
- `OBJECT_NOT_KNOWN (4) (CS0/96)` – A 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)` – The device identifier specified in `device` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

- **Incomplete Names** – For private data versions 2-4, the names returned by this service may not be the full names since the private data confirmation event has a limit of 15 characters for the name.
- **Security** – Communication Manager 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_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 and Later Syntax

```

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

/*
 * attQueryDeviceName() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryDeviceName(
    ATTPrivateData_t *privateData,
    DeviceID_t *device);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTQueryDeviceNameConfEvent - Service Response Private Data */

typedef struct ATTEvent_t
{
    ATTEventType_t eventType; /* ATT_QUERY_DEVICE_NAME_CONF */
    union
    {
        ATTQueryDeviceNameConfEvent_t queryDeviceName;
    } u;
} ATTEvent_t;

typedef struct ATTQueryDeviceNameConfEvent_t {
    ATTDeviceType_t deviceType;
    DeviceID_t device;
    DeviceID_t name; /* 1-27 ASCII
character string */
    ATTUnicodeDeviceID uname; /* name in Unicode */
} ATTQueryDeviceNameConfEvent_t;

typedef enum ATTDeviceType_t
{
    ATT_DT_UNKNOWN = 0,
    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;

typedef struct ATTUnicodeDeviceID {
    unsigned short count;
    short value[64];
} ATTUnicodeDeviceID;

```

Private Data Versions 2-4 Syntax

```

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

RetCode_t attQueryDeviceName(
    ATTPrivateData_t *privateData,
    DeviceID_t *device);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTV4QueryDeviceNameConfEvent - Service Response Private Data */

typedef struct ATTEvent_t
{
    ATTEventType_t eventType; /* ATTV4_QUERY_DEVICE_NAME_CONF */
    union
    {
        ATTV4QueryDeviceNameConfEvent_t v4queryDeviceName;
    } u;
} ATTEvent_t;

typedef struct ATTV4QueryDeviceNameConfEvent_t {
    ATTDeviceType_t deviceType;
    DeviceID_t device;
    char name[16]; /* 1-15 ASCII
character string */
} ATTV4QueryDeviceNameConfEvent_t;

typedef enum ATTDeviceType_t
{
    ATT_DT_UNKNOWN = 0,
    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: `CSTAQueryDndConfEvent`
- 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 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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CSTA_DEVICE_IDENTIFIER` (12) – An invalid device identifier has been specified in device.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The request has failed for one of the following reasons:
 - The Send All Calls feature has not been administered for the specified extension. A partial list of extensions not supporting the Send All Calls feature is:
 - Hunt groups
 - VDNs
 - Announcements
 - Data modules
 - DCS extensions
 - The device identifier specified in `device` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

- Do Not Disturb Event – The [Do Not Disturb Event](#) notifies an application when the Send All Calls feature is activated or deactivated at a monitored station.
- Send All Calls Activation/Deactivation – An application can activate or deactivate the Send Call Calls feature using the [Set Do Not Disturb Feature Service](#).

Syntax

```

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

/* cstaQueryDoNotDisturb() - Service Request */

RetCode_t    cstaQueryDoNotDisturb(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *device,
    PrivateData_t  *privateData);

/* CSTAQueryDndConfEvent - Service Response */

typedef struct
{
    ACSHandle_t    acsHandle;
    EventClass_t   eventClass;           /* CSTACONFIRMATION */
    EventType_t    eventType;           /* CSTA_QUERY_DND_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAQueryDndConfEvent_t queryDnd;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAQueryDndConfEvent_t {
    unsigned char    doNotDisturb;       /* TRUE = on, FALSE = off */
} CSTAQueryDndConfEvent_t;

```

Query Endpoint Registration Info Service (Private Data Version 11 and later)

Summary

- Direction: Client to Switch
- Function: `cstaEscapeService()`
- Confirmation Event: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attQueryEndpointRegistrationInfo()`
- Private Data Confirmation Event: `ATTQueryEndpointRegistrationInfoConfEvent`
- Service Parameters: `noData`
- Private Parameters: `device`
- Ack Parameters: `noData`
- Ack Private Parameters: `device, serviceState, registeredEndpoints`
- Nak Parameter: `universalFailure`

Functional Description:

This service provides the service state of a station extension and a list of H.323 and/or SIP endpoints registered to that station extension.

The Query Endpoint Registration Info service is available beginning with Avaya Communication Manager Release 6.3.2 and AE Services Release 6.3.1.

This service is only available if the TSAPI Link is administered with ASAI Link Version 6 or later and if the application has negotiated private data version 11 or later.

Applications should use the `cstaGetAPICaps()` service and check the value of the `queryEndpointRegistrationInfo` parameter in the private data accompanying the CSTA Get API Caps Confirmation event to determine whether this service is available.

Applications should invoke `cstaMonitorDevice()` for the queried device before invoking this query because Avaya Communication Manager cannot provide the service state of a SIP station that is not monitored.

Note:- Query Endpoint Registration Info Service will not support the increase number of device register to a single extension from 3 to 10 as introduce in AE Services 8.0.1 DMCC

Service Parameters:

None for this service.

Private Parameters:

`device` [mandatory] Must be a valid station extension.

Ack Parameters:

None for this service.

Ack Private Parameters:

`device` [mandatory] The extension number of the station.

`serviceState` [mandatory] The service state of the station extension.
The possible values are:

- `SS_IN_SERVICE` – The station is in service
- `SS_OUT_OF_SERVICE` – The station is out of service.
- `SS_UNKNOWN` – The station's service state is unknown.

`registeredEndpoints` [mandatory] A list structure containing the H.323 and/or SIP endpoints registered to the station extension.
The list structure includes a `count` (0-4) and an array of `ATT-RegisteredEndpointInfo_t` structures.

Each array element includes the following fields:

- `instanceID` – For H.323 endpoints registered through DMCC, the `instanceID` is 0-2. For H.323 endpoints not registered through DMCC and for SIP endpoints, the `instanceID` is always 0. To uniquely identify an endpoint, applications must use both the `endpointAddress` and `instanceID` fields.
- `endpointAddress` – For H.323 endpoints, this is the IP address of the endpoint. For SIP endpoints, this is the empty string (“”).
- `switchEndIpAddress` – The switch-end IP address serving the endpoint.
- `macAddress` – The Media Access Control (MAC) address received from the endpoint when the endpoint registered, or if the endpoint's MAC address is unknown, the value “00:00:00:00:00:00”.
- `productID` – For H.323 endpoints, this is an identifier submitted by the endpoint during registration. Its value is one of the product IDs administered on the Avaya Communication Manager system-parameters customer-options screen. For SIP endpoints, the `productID` is “SIP_Phone”.

- `networkRegion` – The network region (1-250) administered for the endpoint on Avaya Communication Manager.
- `mediaMode` – The media mode in use by the endpoint. The possible values are:
 - `MM_CLIENT_SERVER` – The endpoint is registered in either client media mode or server media mode.
 - `MM_TELECOMMUTER` – The endpoint is registered in Telecommuter media mode.
 - `MM_NONE` – The endpoint is registered without media control. This media mode is sometimes referred to as “Shared Control” because it allows a DMCC application to share control of an extension with another endpoint registered to that extension.
 - `MM_OTHER` – The endpoint is registered with some other media mode not listed above.
- `dependencyMode` – The dependency mode in use by the endpoint. The possible values are:
 - `DM_MAIN` – The endpoint is registered with dependency mode Main. The endpoint can originate and receive calls. Only one endpoint can be registered to the extension with dependency mode Main. Typically, this is a physical set or an IP softphone.
 - `DM_DEPENDENT` – The endpoint is registered with dependency mode Dependent. An endpoint can only register with this dependency mode if another endpoint is already registered with dependency mode Main.
 - `DM_INDEPENDENT` – The endpoint is registered with dependency mode Independent. The endpoint can originate and receive calls even if another endpoint is not registered with dependency mode Main.
 - `DM_OTHER` – The endpoint is registered with some other dependency mode not listed above.
- `unicodeScript` – For H.323 endpoints, this is a set of bit flags indicating which Unicode character sets are supported by the station. For SIP endpoints, this parameter is set to `US_NONE`.
- `stationType` – The station type administered for the extension.
- `signalingProtocol` – The signaling protocol for the endpoint. The possible values are:
 - `SP_H323` – The endpoint registered as an H.323 endpoint.
 - `SP_SIP` – The endpoint registered as a SIP endpoint.
 - `SP_NOT_SPECIFIED` – Avaya Communication Manager cannot provide the endpoint’s signaling protocol.

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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `VALUE_OUT_OF_RANGE (3)` – The service request has failed for one of the following reasons:
 - The stream was not opened with private data version 11 (or later).
 - The TSAPI Link is not administered with ASAI Link Version 6 (or later).
 - Avaya Communication Manager does not support this query.
- `OBJECT_NOT_KNOWN (4)` – The service request did not specify a value for the device parameter.
- `INVALID_CSTA_DEVICE_IDENTIFIER (12)` – An invalid device identifier has been specified in device. For example, the device is not administered on Avaya Communication Manager, or is not a station extension number.

Detailed Information:

- Logical Agents – Logical agent IDs cannot be queried by this service.
- Multiple Registered Endpoints – When multiple H.323 and/or SIP endpoints are registered to the station extension, the query confirmation event includes the information for up to four registered endpoints.
- SIP Endpoint Address – For SIP endpoints, the query does not provide the endpoint; the `endpointAddress` field is set to the empty string (""). The Endpoint Registered Event does provide the endpoint address when a SIP endpoint registers to a monitored station extension.
- SIP Endpoint MAC Address – For SIP endpoints, the query does not provide the MAC address; the `macAddress` field is set to "00:00:00:00:00:00". The Endpoint Registered Event does provide the MAC address when a SIP endpoint registers to a monitored station extension.
- SIP Station Extensions – For SIP station extensions, the query confirmation event is only guaranteed provide the extension's service state if the station extension is being monitored. If the SIP station extension is not being monitored, then the `serviceState` may be reported as `SS_UNKNOWN`.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 11 and Later Syntax

```

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

/*
 * attQueryEndpointRegistrationInfo() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryEndpointRegistrationInfo(
    ATTPrivateData_t *privateData,
    DeviceID_t *device);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/*
 * ATTQueryEndpointRegistrationInfoConfEvent -
 * Service Response Private Data
 */

typedef struct ATTEvent_t
{
    ATTEventType_t eventType;

    /*
    ATT_QUERY_ENDPOINT_REGISTRATION_INFO_CONF */
    union
    {
        ATTQueryEndpointRegistrationInfoConfEvent_t
        queryEndpointRegistrationInfo;
    } u;
} ATTEvent_t;

typedef struct ATTQueryEndpointRegistrationInfoConfEvent_t {
    DeviceID_t device;
    ATTSERVICE_STATE_t serviceState;
    ATTRegisteredEndpointList_t registeredEndpoints;
} ATTQueryEndpointRegistrationInfoConfEvent_t;

typedef enum ATTSERVICE_STATE_t {
    SS_UNKNOWN = -1,
    SS_OUT_OF_SERVICE = 0,
    SS_IN_SERVICE = 1
} ATTSERVICE_STATE_t;

typedef struct ATTRegisteredEndpointList_t {
    unsigned int count;
    ATTRegisteredEndpointInfo_t *endpoint;
} ATTRegisteredEndpointList_t;

```

Query Endpoint Registration Info Service (Private Data Version 11 and later)

```

typedef struct ATTRegisteredEndpointInfo_t {
    ATTEndpointInstanceID_t      instanceID;
    ATTEndpointAddress_t         endpointAddress;
    ATTIPAddress_t               switchEndIpAddress;
    ATTMACAddress_t              macAddress;
    ATTProductID_t               productID;
    short                        networkRegion;
    ATTMediaMode_t               mediaMode;
    ATTDependencyMode_t          dependencyMode;
    ATTUnicodeScript_t           unicodeScript;
    ATTStationType_t             stationType;
    ATTSignalingProtocol_t       signalingProtocol;
} ATTRegisteredEndpointInfo_t;

typedef unsigned short  ATTEndpointInstanceID_t;

typedef char            ATTEndpointAddress_t[256];

typedef char            ATTIPAddress_t[46];

typedef char            ATTMACAddress_t[18];

typedef char            ATTProductID_t[16];

typedef enum ATTMediaMode_t {
    MM_OTHER = -1,
    MM_NONE = 0,
    MM_CLIENT_SERVER = 1,
    MM_TELECOMMUTER = 2
} ATTMediaMode_t;

typedef enum ATTDependencyMode_t {
    DM_OTHER = -1,
    DM_MAIN = 1,
    DM_DEPENDENT = 2,
    DM_INDEPENDENT = 3
} ATTDependencyMode_t;

typedef unsigned int    ATTUnicodeScript_t;

#define US_NONE          0x00000000
#define US_BASIC_LATIN   0x00000001
#define US_LATIN_1_SUPPLEMENT 0x00000002
#define US_LATIN_EXTENDED_A 0x00000004
#define US_LATIN_EXTENDED_B 0x00000008
#define US_GREEK_COPTIC   0x00000010
#define US_CYRILLIC       0x00000020
#define US_ARMENIAN       0x00000040
#define US_HEBREW         0x00000080
#define US_ARABIC         0x00000100

```

```

#define US_DEVANAGARI
    0x00000200
#define US_BENGALI
    0x00000400
#define US_GURMUKHI
    0x00000800
#define US_GUJARATI
    0x00001000
#define US_ORIYA
    0x00002000
#define US_TAMIL
    0x00004000
#define US_TELUGU
    0x00008000
#define US_KANNADA
    0x00010000
#define US_MALAYALAM
    0x00020000
#define US_SINHALA
    0x00040000
#define US_THAI
    0x00080000
#define US_LAO
    0x00100000
#define US_MYANMAR
    0x00200000
#define US_GEORGIAN
    0x00400000
#define US_TAGALOG
    0x00800000
#define US_KHMER
    0x01000000
#define US_HALF_WIDTH_AND_FULL_WIDTH_KANA      0x02000000
#define US_CJK_RADICALS_SUPPLEMENT_JAPAN      0x04000000
#define US_CJK_RADICALS_SUPPLEMENT_CHINA_S    0x08000000
#define US_CJK_RADICALS_SUPPLEMENT_CHINA_T    0x10000000
#define US_CJK_RADICALS_SUPPLEMENT_KOREAN     0x20000000
#define US_CJK_RADICALS_SUPPLEMENT_VIETNAM    0x40000000
#define US_HANGUL_JAMO
    0x80000000

typedef char                                ATTStationType_t[16];

typedef enum ATTSignalingProtocol_t {
    SP_NOT_PROVIDED = -1,
    SP_H323 = 1,
    SP_SIP = 2
} ATTSignalingProtocol_t;

```

Query Forwarding Service

Summary

- Direction: Client to Switch
- Function: `cstaQueryForwarding()`
- Confirmation Event: `CSTAQueryFwdConfEvent`
- 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. For this service, 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 be turned on for many types (Communication Manager redirection criteria), and the actual forward type is dependent on how the feature is administered in Communication Manager.

Service Parameters:

<code>device</code>	[mandatory] Must be a valid on-PBX station extension that supports the Call Forwarding feature.

Ack Parameters:

<code>forward</code>	[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: <code>forwardingType</code> , <code>forwardingOn</code> , and <code>forwardDN</code> . For Communication Manager, <code>forwardingType</code> will always be <code>FWD_IMMEDIATE</code> ; <code>forwardingOn</code> will indicate the feature status (<code>TRUE</code> indicates on, <code>FALSE</code> indicates off); and <code>forwardDN</code> will contain the forward-to-number.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The <code>error</code> parameter in this event may contain the following error value, or one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997. • <code>INVALID_CSTA_DEVICE_IDENTIFIER</code> (12) – An invalid device identifier has been specified in <code>device</code>. • <code>GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY</code> (41) – The device identifier specified in <code>device</code> corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

- Call Forwarding Immediate Activation/Deactivation – An application can activate or deactivate the Call Forwarding Immediate feature using the [Set Forwarding Feature Service](#).
- Forwarding Event – The [Forwarding Event](#) notifies an application when the Call Forwarding Immediate feature is activated or deactivated at a monitored station.
- Forwarding Types – 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_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *deviceID,
    PrivateData_t  *privateData);

/* CSTAQueryFwdConfEvent - Service Response */

typedef struct
{
    ACSHandle_t    acsHandle;
    EventClass_t   eventClass;           /* CSTACONFIRMATION */
    EventType_t    eventType;           /* CSTA_QUERY_FWD_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAQueryFwdConfEvent_t  queryFwd;
            }u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAQueryFwdConfEvent_t {
    ListForwardParameters_t forward;
} CSTAQueryFwdConfEvent_t;

typedef struct ListForwardParameters_t {
    unsigned short    count;           /* will be at most 1 */
} ListForwardParameters_t;

typedef struct ForwardingInfo_t {
    ForwardingType_t  forwardingType; /* FWD_IMMEDIATE */

```

```

        unsigned char        forwardingOn;           /* TRUE = on,
FALSE = off */
        DeviceID_t           forwardDN;
    } ForwardingInfo_t;

typedef enum ForwardingType_t {
    FWD_IMMEDIATE = 0,        /* this is the 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 Indicator 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 Indicator Service provides status of the message waiting indicator for a device expressed as on or off. The applications that turn the indicator on (that is, ASAI, Property Management, Message Center, Voice Processing, and Leave Word Calling) are reported in the private data.

Service Parameters:

<code>device</code>	[mandatory] Must be a valid on-PBX station extension that supports the Message Waiting Indicator (MWI) feature.

Ack Parameters:

<code>messages</code>	[mandatory] Indicates the on/off status (<code>TRUE</code> indicates on, <code>FALSE</code> indicates off) of the MWI for this device.

Ack Private Parameters:

<code>applicationType</code>	[mandatory] Indicates the applications that turned on the MWI for the device

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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CSTA_DEVICE_IDENTIFIER` (12) – An invalid device identifier has been specified in `device`.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `device` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

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 (i.e., DLG, CVLAN, or TSAPI)	LWC	PMS	Voice	MCS

- **Message Waiting Event** – The [Private Data Version](#) 14

If private data accompanies a `CSTALoggedInEvent`, 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 `CSTALoggedInEvent` 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>
```

```
/* ATTV14LoggedInEvent - CSTA Unsolicited Event Private Data */
```

```
typedef struct
{
```

```

    ATTEventType_t    eventType;          /* ATT_LOGGED_ON */
    union
    {
        ATTV14LoggedOnEvent_t    loggedOnEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV14LoggedOnEvent_t {
    ATTWorkMode_t    workMode;
} ATTV14LoggedOnEvent_t;

typedef enum ATTWorkMode_t {
    WM_NONE = -1,
    WM_AUX_WORK = 1,
    WM_AFTCAL_WK = 2,
    WM_AUTO_IN = 3,
    WM_MANUAL_IN = 4
} ATTWorkMode_t;

```

- Message Waiting Event notifies an application when a monitored station's message waiting indicator is turned on or off.
- 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_t    acsHandle,
    InvokeID_t     invokeID,
    DeviceID_t     *device,
    PrivateData_t  *privateData);

/* CSTAQueryMwiConfEvent - Service Response */

typedef struct
{
    ACSHandle_t    acsHandle;
    EventClass_t   eventClass;           /* CSTACONFIRMATION */
    EventType_t    eventType;           /* CSTA_QUERY_MWI_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAQueryMwiConfEvent_t queryMwi;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAQueryMwiConfEvent_t {
    unsigned char    messages;           /* TRUE = on, FALSE = off */
} CSTAQueryMwiConfEvent_t;

```

Private Data Syntax

```

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

/* ATTQueryMwiConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_QUERY_MWI_CONF */
    union
    {
        ATTQueryMwiConfEvent_t  queryMwi;
    }u;
} ATTEvent_t;

typedef struct ATTQueryMwiConfEvent_t
{
    ATTMwiApplication_t    applicationType;
} ATTQueryMwiConfEvent_t;

typedef unsigned char    ATTMwiApplication_t;

#define AT_MCS                0x01        /* bit 1 */
#define AT_VOICE              0x02        /* bit 2 */
#define AT_PROPMGT            0x04        /* 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: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attV10QueryStationStatus()` (private data version 10 and later), `attQueryStationStatus()` (private data versions 2 and later)
- Private Data Confirmation Event: `ATTQueryStationStatusConfEvent` (private data version 10 and later), `ATTV9QueryStationStatusConfEvent` (private data versions 2-9)
- Service Parameters: `noData`
- Private Parameters: `device`
- Ack Parameters: `noData`
- Ack Private Parameters: `stationStatus, serviceState`
- Nak Parameter: `universalFailure`

Functional Description:

The Query Station Status service provides state of a station: idle or busy. The "busy" state is returned if the station is active with a call or out of service. The "idle" state is returned if the station is in service and not active with any call.

Private data version 10 supports a new version of this service, `attV10QueryStationStatus()`. It is similar to the pre-existing `attQueryStationStatus()` service, but its confirmation event also includes the station's service state (in-service or out-of-service), if available:

To receive a service state other than `SS_UNKNOWN` in the confirmation event:

- Avaya Communication Manager must be running Release 6.2 or later;
- The TSAPI CTI Link must be administered with ASAI Link Version 5 or later; and
- The station extension being queried must not correspond to a SIP endpoint.

NOTE:

Avaya Communication Manager Release 6.2 does not provide the service state of SIP endpoints. Beginning with Avaya Communication Manager Release 6.3, the service state is provided for monitored SIP endpoints.

Service Parameters:

None for this service.

Private Parameters:

`device` [mandatory] Must be a valid station device.

Ack Parameters:

None for this service.

Ack Private Parameters:

`stationStatus` [mandatory] Specifies the busy/idle state (`TRUE` indicates busy, `FALSE` indicates idle) of the station.

`serviceState` [optional – partially supported] Indicates the service state of the station, if available.

Possible values are:

- `SS_IN_SERVICE` – The station is in service
- `SS_OUT_OF_SERVICE` – The station is out of service.
- `SS_UNKNOWN` – The station's service state is unknown.

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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CSTA_DEVICE_IDENTIFIER` (12) – An invalid device identifier has been specified in `device`.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `device` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

None for this service.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 10 and Later Syntax

```

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

/*
 * attV10QueryStationStatus() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV10QueryStationStatus(
    ATTPrivateData_t *privateData,
    DeviceID_t *device);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTQueryStationStatusConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t eventType; /* ATT_QUERY_STATION_STATUS_CONF
*/
    union
    {
        ATTQueryStationStatusConfEvent_t queryStationStatus;
    } u;
} ATTEvent_t;

typedef struct ATTQueryStationStatusConfEvent_t {
    unsigned char stationStatus; /* TRUE = busy, FALSE
= idle */
    ATTSERVICEState_t serviceState;
} ATTQueryStationStatusConfEvent_t;

typedef enum ATTSERVICEState_t {
    SS_UNKNOWN = -1,
    SS_OUT_OF_SERVICE = 0,
    SS_IN_SERVICE = 1
} ATTSERVICEState_t;

```

Private Data Versions 2-9 Syntax

```

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

/*
 * attQueryStationStatus() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryStationStatus(
    ATTPrivateData_t *privateData,
    DeviceID_t *device);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTV9QueryStationStatusConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t eventType; /*
ATTV9_QUERY_STATION_STATUS_CONF */
    union
    {
        ATTV9QueryStationStatusConfEvent_t v9queryStationStatus;
    } u;
} ATTEvent_t;

typedef struct ATTV9QueryStationStatusConfEvent_t {
    unsigned char stationStatus; /* TRUE = busy, FALSE
= idle */
} ATTV9QueryStationStatusConfEvent_t;

```

Query Time of Day Service

Summary

- Direction: Client to Switch
- Function: `cstaEscapeService()`
- Confirmation Event: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attQueryTimeOfDay()`
- Private Data Confirmation Event: `ATTQueryTodConfEvent`
- 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:

None for this service.

Ack Parameters:

None for this service.

Ack Private Parameters:

<code>time</code>	[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 Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code> . The <code>error</code> parameter in this event may contain one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997.

Detailed Information:

None for this service.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 Syntax

```

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

/*
 * attQueryTimeOfDay() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryTimeOfDay(
    ATTPrivateData_t *privateData);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTQueryTodConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t eventType; /* ATT_QUERY_TOD_CONF */
    union
    {
        ATTQueryTodConfEvent_t queryTOD;
    } u;
} ATTEvent_t;

typedef struct ATTQueryTodConfEvent_t {
    short year;
    short month;
    short day;
    short hour;
    short minute;
    short second;
} ATTQueryTODConfEvent_t;

```

Query Trunk Group Service

Summary

- Direction: Client to Switch
- Function: `cstaEscapeService()`
- Confirmation Event: `CSTAEscapeSvcConfEvent`
- Private Data Function: `attQueryTrunkGroup()`
- Private Data Confirmation Event: `ATTQueryTgConfEvent`
- 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:

None for this service.

Private Parameters:

<code>device</code>	[mandatory] Specifies a valid trunk group access code.

Ack Parameters:

None for this service.

Ack Private Parameters:

<code>idleTrunks</code>	[mandatory] The number of "idle" trunks in the group.
<code>usedTrunks</code>	[mandatory] The number of "in use" trunks in the group

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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CSTA_DEVICE_IDENTIFIER (12)` – An invalid device identifier has been specified in device.

Detailed Information:

None for this service.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 Syntax

```

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

/*
 * attQueryTrunkGroup() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attQueryTrunkGroup(
    ATTPrivateData_t *privateData,
    DeviceID_t *device);

typedef struct ATTPrivateData_t {
    char vendor[32];
    unsigned short length;
    char data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTQueryTgConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t eventType; /* ATT_QUERY_TG_CONF */
    union
    {
        ATTQueryTgConfEvent_t queryTg;
    } u;
} 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: `CSTAEscapeSvcConfEvent`
- 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 performed anytime during the life of a call.

Service Parameters:

None for this service.

Private Parameters:

<code>call</code>	[mandatory] Specifies the normal <code>callID</code> of a call. This is a Connection Identifier. The <code>deviceID</code> is ignored.

Ack Parameters:

None for this service.

Ack Private Parameters:

<code>ucid</code>	[mandatory] Specifies the Universal Call ID (<code>UCID</code>) of the requested call. The <code>UCID</code> is a unique call identifier across switches and the network. A valid <code>UCID</code> is a null-terminated ASCII character string. If there is no <code>UCID</code> associated with this call, the <code>ucid</code> contains the <code>ATT_NULL_UCID</code> (a 20-character string of all zeros).

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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.

Detailed Information:

None for this service.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 Syntax

```

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

/* attQueryUCID() - Service Request Private Data Formatting Function
*/

RetCode_t attQueryUCID(
    ATTPrivateData_t      *privateData,
    ConnectionID_t        *call);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTQueryUcidConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t      eventType;          /* ATT_QUERY_UCID_CONF */
    union
    {
        ATTQueryUcidConfEvent_t queryUCID;
    } u;
} ATTEvent_t;

typedef char            ATTUCID_t[64];

typedef struct ATTQueryUcidConfEvent_t {
    ATTUCID_t           ucid;
} ATTQueryUcidConfEvent_t;

```

Chapter 9: Snapshot Service Group

The Snapshot Service Group provides services that enable the client to get information about a particular call, or information about calls associated with a particular device.

The following sections describe the Snapshot services supported by AE Services:

- [Snapshot Call Service](#) on page 549
- [Snapshot Device Service](#) on page 556

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:

<code>snapshotObj</code>	[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.

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 included in the `snapshotData` as well as the following detailed information for each endpoint: Device ID, Call ID, and Local Connection State of the call at the device.

In some cases, the `snapshotData` may not contain all of the device endpoints that are associated with the call. (For example, when the call is alerting at a large number of bridged appearances.) Typically, the `snapshotData` will contain no more than 20 device endpoints.

Ack 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, or when a device clears from a call).

Conceptually, the `deviceHistory` parameter provides a list of entries, where each entry contains information about a `DeviceID` that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, this list contains at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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).

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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
            }
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASnapshotCallConfEvent_t {
    CSTASnapshotCallData_t snapshotCall;
} CSTASnapshotCallConfEvent_t;

typedef struct CSTASnapshotCallData_t {
    unsigned int count;
    /* call count */
    CSTASnapshotCallResponseInfo_t *info;
} CSTASnapshotCallData_t;

```

```
typedef struct CSTASnapshotCallResponseInfo_t {
    SubjectDeviceID_t          deviceOnCall;
    ConnectionID_t             callIdentifier;
    LocalConnectionState_t     localConnectionState;
} CSTASnapshotCallResponseInfo_t;

typedef enum LocalConnectionState_t {
    CS_NONE = -1,
    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 7 and Later Syntax

The CSTA Snapshot Call Confirmation event includes a private data, an *ATTSnapshotCallConfEvent*, for private data version 7 and later. The *ATTSnapshotCallConfEvent* provides the *deviceHistory* private data parameter.

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

/* ATTSnapshotCallConfEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_SNAPSHOT_CALL_CONF */
    union
    {
        ATTSnapshotCallConfEvent_t    snapshotCallConf;
    } u;
} ATTEvent_t;

typedef struct ATTSnapshotCallConfEvent_t {
    DeviceHistory_t    deviceHistory;
} ATTSnapshotCallConfEvent_t;

typedef struct DeviceHistory_t {
    unsigned int                count;          /* at most 1 */
    DeviceHistoryEntry_t        *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t                olddeviceID;
    CSTAEventCause_t          cause;
    ConnectionID_t            oldconnectionID;
} DeviceHistoryEntry_t;
```

Snapshot Device Service

Summary

- Direction: Client to Switch
- Function: `cstaSnapshotDeviceReq()`
- Confirmation Event: `CSTASnapshotDeviceConfEvent`
- Private Data Confirmation Event: `ATTSnapshotDeviceConfEvent` (private data version 5 and later), `ATTV4SnapshotDeviceConfEvent` (private data versions 2-4)
- Service Parameters: `snapshotObj`
- Ack Parameters: `snapshotDevice`
- Ack Private Parameters: `pSnapshotDevice`
- 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.

Service Parameters:

`snapshotObj` [mandatory] Must be a valid station extension number.

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:

`pSnapshotDevice` [mandatory] Contains a sequence of information about each call on the device. Information for each call includes the `connectionID` and the Communication Manager call state for each call at the snapshot device.

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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors - universalFailure](#) on page 997.

- `INVALID_CSTA_DEVICE_IDENTIFIER` (12) – An invalid device identifier has been specified in `snapshotObj`.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41) – The device identifier specified in `snapshotObj` corresponds to a SIP station and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.

Detailed Information:

- The ECMA-180 definition for the confirmation event 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:

Communication Manager 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 specification. 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,
    PrivateData_t  *privateData);

/* CSTASnapshotDeviceConfEvent - 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;
} CSTAEvent_t;

typedef struct CSTASnapshotDeviceConfEvent_t {
    CSTASnapshotDeviceData_t    snapshotData;
} CSTASnapshotDeviceConfEvent_t;

typedef struct CSTASnapshotDeviceData_t {
    unsigned int
    count;
    CSTASnapshotDeviceResponseInfo_t    *info;
} CSTASnapshotDeviceData_t;

typedef struct CSTASnapshotDeviceResponseInfo_t {

```

```

        ConnectionID_t          callIdentifier;
        CSTACallState_t         localCallState;
    } CSTASnapshotDeviceResponseInfo_t;

typedef struct CSTACallState_t {
    unsigned int                count;          /* number 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 and Later 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 {
    unsigned 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;

```

Private Data Version 2-4 Syntax

```

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

/* ATTV4SnapshotDeviceConfEvent - Service Response Private Data */

typedef struct ATTEvent_t
{
    ATTEventType_t    eventType; /* ATTV4_SNAPSHOT_DEVICE_CONF */
    union
    {
        ATTV4SnapshotDeviceConfEvent_t    v4snapshotDevice;
    } u;
} ATTEvent_t;

typedef struct ATTV4SnapshotDeviceConfEvent_t {
    unsigned short    count;
    ATTSnapshotDevice_t    *snapshotDevice[6];
} ATTV4SnapshotDeviceConfEvent_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

The Monitor Service Group provides services to monitor calls and devices. This chapter includes the following sections:

- [Overview](#) on page 564
- [Change Monitor Filter Service](#) on page 570
- [Monitor Call Service](#) on page 578
- [Monitor Calls Via Device Service](#) on page 591
- [Monitor Device Service](#) on page 603
- [Monitor Ended Event Report](#) on page 614
- [Monitor Stop On Call Service \(Private\)](#) on page 616
- [Monitor Stop Service](#) on page 620

Overview

This overview provides a high level description of each of the monitor services that AE Services supports. Additionally, it includes the following topics:

- [Event Filters and Monitor Services](#) on page 565
- [The localConnectionInfo Parameter for Monitor Services](#) on page 569

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 service⁴ 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. Event reports for calls that were already present at the VDN or ACD prior to the monitor request are not provided. 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 service⁵ 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

⁴ The Monitor Calls Via Device Service is the call-type Monitor Start Service on a static device identifier in ECMA-179.

⁵ The Monitor Device Service is the device-type Monitor Start Service on a static device identifier in ECMA-179.

monitor request and arrive at the device after the monitor request is acknowledged. If a call is dropped, forwarded, or transferred from the device, and the device has ceased to participate in the call, no further events of the call are reported.

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.

Monitor Stop Service — cstaMonitorStop()

An application uses this service to cancel a previously requested Monitor Service.

Event Filters and Monitor Services

[Table 14](#) shows the relationship between event filters and monitor services.

- A value of "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.
- A value of "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 (i.e., Advice of Charge, Entered Digits, Endpoint Registered, and Endpoint Unregistered) 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					

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)
Advice of Charge (private data version 5 and later)	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
Endpoint Registered (private data version 11 and later)	On	On/Off ⁶	On	On	On
Endpoint Unregistered (private data version 11 and later)	On	On/Off	On	On	On
Established	On/Off	On/Off	On	On	On/Off
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	On/Off	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

⁶ For Communication Manager releases earlier than 6.3.2, or when the TSAPI CTI link is administered with an ASAI link version less than 6, the Endpoint Registered and Endpoint Unregistered events are always filtered (On).

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)
Transferred	On/Off	On/Off	On	On	On/Off

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)
Agent Event Filters					
Logged On	On	On	On/Off	On	On
Logged Off	On	On	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/Off ⁷	On	On	On
Forwarding	On	On/Off	On	On	On
Message Waiting	On	On/Off ⁸	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

⁷ For Communication Manager releases earlier than 5.0, or when the TSAPI CTI link is administered with an ASAI link version less than 5, the Do Not Disturb and Forwarding events are always filtered (On).

⁸ For Communication Manager releases earlier than 6.3.2, or when the TSAPI CTI link is administered with an ASAI link version less than 5, the Message Waiting event is 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 and later), `attMonitorFilter()` (private data versions 2-4)
- Private Data Confirmation Event: `ATTMonitorConfEvent` (private data version 5 and later), `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 for a previously established monitor association.

Service Parameters:

<code>monitorCrossRefID</code>	[mandatory] Must be a valid Cross Reference ID that was returned in a previous <code>CSTAMonitorConfEvent</code> of this <code>acsOpenStream</code> session.
<code>filterList</code>	[mandatory — partially supported] Specifies the filters to be changed. Call Filter, Agent Filter, Feature Filter, and Private Filter are supported. Setting a filter for an event (for example, <code>CF_CALL_CLEARED=0x8000</code>) in the <code>monitorFilter</code> means that the event will be filtered out and no such event reports will be sent to the application on that monitor. 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 Maintenance Filter is not supported. If present, it will be ignored.

Private Parameters:

`privateFilter` [optional] Specifies the Communication Manager private filters to be changed. The following private filters are supported:

- Private data version 11 and later:
 - `ATT_ENTERED_DIGITS_FILTER`
 - `ATT_CHARGE_ADVICE_FILTER`
 - `ATT_ENDPOINT_UNREGISTERED_FILTER`
 - `ATT_ENDPOINT_REGISTERED_FILTER`
- Private data versions 5-10:
 - `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 the `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 Communication Manager 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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 877.

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;

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                     0x10
#define AF_WORK_NOT_READY             0x08
#define AF_WORK_READY                0x04      /* not
supported */

typedef unsigned char    CSTAMaintenanceFilter_t

#define MF_BACK_IN_SERVICE            0x80      /* not
supported */
#define MF_OUT_OF_SERVICE             0x40      /* not
supported */

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 11 and Later Syntax

```

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

/*
 * attMonitorFilterExt() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorFilterExt(
    ATTPrivateData_t      *privateData,
    ATTPrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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
#define ATT_ENDPOINT_UNREGISTERED_FILTER    0x20
#define ATT_ENDPOINT_REGISTERED_FILTER      0x10

/* 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 5-10 Syntax

```

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

/*
 * attMonitorFilterExt() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorFilterExt(
    ATTPrivateData_t      *privateData,
    ATTPrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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
 * Formatting Function
 */

RetCode_t attMonitorFilter(
    ATTPrivateData_t      *privateData,
    ATTV4PrivateFilter_t  privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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 Call Service

Summary

- Direction: Client to Switch
- Function: `cstaMonitorCall()`
- Confirmation Event: `CSTAMonitorConfEvent`
- Private Data Function: `attMonitorFilterExt()` (private data version 5 and later), `attMonitorFilter()` (private data versions 2-4)
- Private Data Confirmation Event: `ATTMonitorCallConfEvent` (private data version 5 and later), `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` for a snapshot 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:

<code>call</code>	[mandatory] <code>ConnectionID</code> of the call to be monitored.
<code>monitorFilter</code>	[optional – partially supported] Specifies the filters to be used with <code>call</code>. 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 for an event (for example, <code>CF_CALL_CLEARED=0x8000</code>) in the <code>monitorFilter</code> means that the event will be filtered out and no such event reports will be sent to the application on that monitor. 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:

<code>privateFilter</code>	[optional] Specifies the Communication Manager private filters to be changed. The following private filters are supported: • Private data version 11 and later: – <code>ATT_ENTERED_DIGITS_FILTER</code> – <code>ATT_CHARGE_ADVICE_FILTER</code> – <code>ATT_ENDPOINT_UNREGISTERED_FILTER</code> – <code>ATT_ENDPOINT_REGISTERED_FILTER</code> • Private data versions 5-10: – <code>ATT_ENTERED_DIGITS_FILTER</code> – <code>ATT_CHARGE_ADVICE_FILTER</code> • Private data versions 2-4: – <code>ATT_V4_ENTERED_DIGITS_FILTER</code>

See [Table 14](#) to determine which filters are under the control of the application, that is, can be turned on and off.

Ack Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle chosen by the TSAPI Service for the monitor. This handle is a unique value within an <code>acsOpenStream</code> 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 a Monitor Stop service request to the original <code>cstaMonitorCall()</code> request.
<code>monitorFilter</code>	[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 <code>monitorFilter</code> 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 <code>monitorFilter</code>. 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:

<code>usedFilter</code>	[optional] Specifies the Communication Manager Private Filter and Event Reports that are to be filtered out on the object being monitored by the application.
<code>snapshotCall</code>	[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 <code>CSTASnapshotCallConfEvent</code> 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CONNECTION_ID_FOR_ACTIVE_CALL` (23) (CS0/100) – The call identifier is outside the range of valid call identifier values.
- `NO_ACTIVE_CALL` (24) (CS3/86) – The application has sent an invalid call identifier. The call does not exist or has been cleared.
- `RESOURCE_BUSY` (33) – The 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 being monitored by another AE Services server. The request cannot be executed because the system limit is exceeded for the maximum number of monitors on a call by CTI applications.

Detailed Information:

See also [Event Report Detailed Information](#) on page 877.

- **Monitor Ended Event Report** — When the monitored call is ended before a `cstaMonitorStop()` request 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 AE Services Servers** — Multiple TSAPI applications may monitor the same call if all of the applications have opened streams to the same AE Services server. However, a `cstaMonitorCall()` request will fail if the call is already being monitored by a TSAPI application on a stream to a different AE Services server, or if the call is already being monitored by a DLG or CVLAN application.
- **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 version 5 and later)** — The `ATTChargeAdviceEvent` is provided, by an outside service, to streams which have enabled Advice of Charge using the `attSetAdviceOfCharge()` escape service. 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
                                         * filters 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;

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                      0x10
#define AF_WORK_NOT_READY             0x08
#define AF_WORK_READY                 0x04      /* not
supported */

typedef unsigned char   CSTAMaintenanceFilter_t

#define MF_BACK_IN_SERVICE            0x80      /* not
supported */
#define MF_OUT_OF_SERVICE             0x40      /* not
supported */

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 11 and Later Syntax

```

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

/*
 * attMonitorFilterExt() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorFilterExt(
    ATTPrivateData_t      *privateData,
    ATTPrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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
#define ATT_ENDPOINT_UNREGISTERED_FILTER    0x20
#define ATT_ENDPOINT_REGISTERED_FILTER      0x10

/* 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 5-10 Syntax

```

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

/*
 * attMonitorFilterExt() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorFilterExt(
    ATTPrivateData_t      *privateData,
    ATTPrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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 {
    unsigned int            count;
    CSTASnapshotCallResponseInfo_t      *pInfo;
} ATTSnapshotCall_t;

typedef struct CSTASnapshotCallResponseInfo_t {
    SubjectDeviceID_t       deviceOnCall;
    ConnectionID_t          callIdentifier;
}

```

```
        LocalConnectionState_t      localConnectionState;
    } CSTASnapshotCallResponseInfo_t;

typedef enum LocalConnectionState_t {
    CS_NONE = -1,
    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 Versions 2-4 Syntax

```

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

/*
 * attMonitorFilter() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorFilter(
    ATTPrivateData_t      *privateData,
    ATTV4PrivateFilter_t  privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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;

#define ATT_MAX_PARTIES_ON_CALL 6

typedef struct ATTV4SnapshotCall_t {
    unsigned 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;
} CSTASnapshotCallResponseInfo_t;

typedef enum LocalConnectionState_t {
    CS_NONE = -1,
    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;
```

Monitor Calls Via Device Service

Summary

- Direction: Client to Switch
- Function: `cstaMonitorCallsViaDevice()`
- Confirmation Event: `CSTAMonitorConfEvent`
- Private Data Function: `attMonitorCallsViaDevice()` (private data version 7 and later), `attMonitorFilterExt()` (private data version 5 and later), `attMonitorFilter()` (private data version 2 and later)
- 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 traditional ACD Split devices, but not station devices. Use `cstaMonitorDevice()` service to monitor stations. This service does not support ACD Split devices when the Expert Agent Selection (EAS) feature is enabled.

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:

<code>deviceID</code>	[mandatory] A valid on-PBX VDN or ACD Split extension to be monitored. A station extension is invalid.
<code>monitorFilter</code>	[optional — partially supported] Specifies the filters to be used with <code>deviceID</code>. 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 for an event (for example, <code>CF_CALL_CLEARED=0x8000</code>) in the <code>monitorFilter</code> means that the event will be filtered out and no such event reports will be sent to the application on the monitor. 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 for this service. If one of these is present, it will be ignored.

Private Parameters:

<code>privateFilter</code>	[optional] Specifies the Communication Manager private filters to be changed. The following private filters are supported: • Private data version 11 and later: – <code>ATT_ENTERED_DIGITS_FILTER</code> – <code>ATT_CHARGE_ADVICE_FILTER</code> – <code>ATT_ENDPOINT_UNREGISTERED_FILTER</code> – <code>ATT_ENDPOINT_REGISTERED_FILTER</code> • Private data versions 5-10: – <code>ATT_ENTERED_DIGITS_FILTER</code> – <code>ATT_CHARGE_ADVICE_FILTER</code> • Private data versions 2-4: – <code>ATT_V4_ENTERED_DIGITS_FILTER</code> See Table 14 to determine which filters are under the control of the application, that is, can be turned on and off.

<code>flowPredictiveCallEvents</code>	[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 <code>CSTADeliveredEvent</code> indicating that the call arrived at the VDN. When the application specifies this parameter as <code>FALSE</code>, this behavior is unchanged. When the application specifies this parameter as <code>TRUE</code>, the monitor also receives events for the outbound call to the <code>calledDevice</code> in the <code>cstaMakePredictiveCall()</code> request. These events may include: • a <code>CSTADeliveredEvent</code> indicating that the call arrived at the <code>calledDevice</code> • a <code>CSTAEstablishedEvent</code> indicating that the call has been answered by the <code>calledDevice</code>; or • a <code>CSTAConnectionClearedEvent</code> indicating that the connection has been cleared at the <code>calledDevice</code>.

Ack Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle chosen by the TSAPI Service for the monitor. This handle is a unique value within an <code>acsOpenStream</code> 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 a Monitor Stop service request to the original <code>cstaMonitorCallsViaDevice()</code> request.
<code>monitorFilter</code>	[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 <code>monitorFilter</code> 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 <code>monitorFilter</code>. Only Call Filter and Call Event Reports are supported. All event reports in Agent Filter, Feature Filter, and Maintenance Filter are set to "On", meaning that there are no reports supported for these events.

Ack Private Parameters:

<code>usedFilter</code>	[optional] Specifies the Communication Manager 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `REQUEST_INCOMPATIBLE_WITH_OBJECT` (2) – The service request has failed for one of the following reasons:
 - The monitored object is not administered correctly in the switch.
 - The monitored object is an adjunct-controlled ACD split or a vector-controlled ACD split.
- `INVALID_CSTA_DEVICE_IDENTIFIER` (12) – An invalid device identifier or extension is specified in `deviceId`.
- `RESOURCE_BUSY` (33) – The 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 877.

- 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 no Diverted Event Report is provided. Subsequent event reports of the call continue.
 - If a call monitored by a `cstaMonitorCallsViaDevice()` request at an ACD or VDN device (A) and is diverted to an ACD or VDN device (B) monitored by another `cstaMonitorCallsViaDevice()` request, then a Diverted Event Report is provided on the monitor for the first ACD or VDN device (A) to indicate that the call has left the domain of the monitored device, and that 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 and subsequent event reports of the call continue to be provided to that monitor. If the resulting call is the one not being monitored, a Conferenced/Transferred Event Report with a new `callID` is sent to the monitor. A Call Ended Event Report for the abandoned call is also sent to the monitor. Subsequent event reports of the new call continue to be sent to the monitor. 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
a VDN or

        * an ACD split */
    CSTAMonitorFilter_t    *monitorFilter,    /* supports call

        * filters 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;

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                      0x10
#define AF_WORK_NOT_READY             0x08
#define AF_WORK_READY                 0x04      /* not
supported */

typedef unsigned char   CSTAMaintenanceFilter_t

#define MF_BACK_IN_SERVICE            0x80      /* not
supported */
#define MF_OUT_OF_SERVICE             0x40      /* not
supported */

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 11 and Later 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];
    unsigned short      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
#define ATT_ENDPOINT_UNREGISTERED_FILTER    0x20
#define ATT_ENDPOINT_REGISTERED_FILTER      0x10

/* 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 7-10 Syntax

```

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

/*
 * attMonitorCallsViaDevice() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorCallsViaDevice(
    ATTPriateData_t      *privateData,
    ATTPriateFilter_t    privateFilter,
    Boolean               flowPredictiveCallEvents);

typedef struct ATTPriateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPriateData_t;

typedef unsigned char  ATTPriateFilter_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 {
    ATTPriateFilter_t      usedFilter;
} ATTMonitorConfEvent_t;

```

Private Data Version 5 and 6 Syntax

```

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

/*
 * attMonitorFilterExt() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorFilterExt(
    ATTPrivateData_t      *privateData,
    ATTPrivateFilter_t     privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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
 * Formatting Function
 */

RetCode_t attMonitorFilter(
    ATTPrivateData_t      *privateData,
    ATTV4PrivateFilter_t  privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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 and later), `attMonitorFilter()` (private data versions 2-4)
- Private Data Confirmation Event: `ATTMonitorConfEvent` (private data version 5 and later), `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 and provides partial support for Agent Event Reports for ACD Split devices.

Beginning with Communication Manager Release 5.0, this service provides partial support for Feature Event Reports for station devices, provided that the TSAPI CTI link is administered with ASAI link version 5 or later.

Beginning with Communication Manager Release 6.3.2, this service provides support for the Endpoint Registered and Endpoint Unregistered private status events for station devices, provided that the TSAPI CTI link is administered with ASAI link version 6 or later and that the application has negotiated private data version 11 or later.

Maintenance Event Reports are not supported.

 NOTE:

Communication Manager 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 336.) If the Charge Advice Event feature is turned on, a trunk group monitored by `cstaMonitorDevice()`, a station monitored by `cstaMonitorDevice()`, or a call monitored by `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:

<code>deviceID</code>	[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 monitor will receive the Charge Advice Event only. It will not receive any other call events.
<code>monitorFilter</code>	[optional — partially supported] Specifies the filters to be used with <code>deviceID</code>. Call Filter/Event Reports are supported for station devices. If a Call Filter is not present, it defaults to no filter, meaning that all Communication Manager CSTA Call Event Reports will be reported. The Agent Filter is supported for ACD Split devices. Setting a filter for an event (for example, <code>CF_CALL_CLEARED=0x8000</code>) in the <code>monitorFilter</code> means that the event will be filtered out and no such event reports will be sent to the application on that monitor. 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:

<code>privateFilter</code>	[optional] Specifies the Communication Manager private filters to be changed. The following private filters are supported: • Private data version 11 and later: – <code>ATT_ENTERED_DIGITS_FILTER</code> – <code>ATT_CHARGE_ADVICE_FILTER</code> – <code>ATT_ENDPOINT_UNREGISTERED_FILTER</code> – <code>ATT_ENDPOINT_REGISTERED_FILTER</code> • Private data versions 5-10: – <code>ATT_ENTERED_DIGITS_FILTER</code> – <code>ATT_CHARGE_ADVICE_FILTER</code> • Private data versions 2-4: – <code>ATT_V4_ENTERED_DIGITS_FILTER</code> See Table 14 to determine which filters are under the control of the application, that is, can be turned on and off.

Ack Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle chosen by the TSAPI Service for the monitor. This handle is a unique value within an <code>acsOpenStream</code> 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 a Monitor Stop service request to the original Monitor Device service request.
<code>monitorFilter</code>	[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 <code>monitorFilter</code> 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 <code>monitorFilter</code>. Maintenance Filters are set to "On", meaning that there are no reports supported for these events.

Ack Private Parameters:

<code>usedFilter</code>	[optional] Specifies the Communication Manager 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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CSTA_DEVICE_IDENTIFIER` (12) – An invalid device identifier or extension is specified in `deviceID`.
- `RESOURCE_BUSY` (33) – The TSAPI Service is busy processing a `cstaMonitorDevice()` service request on the same device. Try again.
- `GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY` (41)

The request has failed for one of the following reasons:

- A required feature has not been provisioned or enabled. For example:
 - The `deviceID` corresponds to a station with type “CTI” and the “CTI Stations” feature is not enabled on Communication Manager.
 - The `deviceID` corresponds to a non-CTI station administered without hardware (AWOH) and the “Phantom Calls” feature is not enabled on Communication Manager.
 - The `deviceID` corresponds to a SIP station, and the “Type of 3PCC Enabled” administered for the station is not set to “Avaya”.
 - The device ID corresponds to a SIP station, and there is an off-pbx-telephone station-mapping administered for `deviceID` on Communication Manager.
- The TSAPI Service could not acquire the license(s) needed to satisfy the request.
- `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 877.

- ACD split — An ACD split can be monitored by this service. The monitor will only receive 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 be monitored with the 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. Agent login IDs are not valid monitor objects.
- Multiple Requests — Multiple applications can have multiple `cstaMonitorDevice()` requests for the same device. An application can have more than one `cstaMonitorDevice()` request on one device. 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);

/* 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;

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                         0x10
#define AF_WORK_NOT_READY                0x08
#define AF_WORK_READY                    0x04          /* not
supported */

typedef unsigned char   CSTAMaintenanceFilter_t

#define MF_BACK_IN_SERVICE               0x80          /* not
supported */
#define MF_OUT_OF_SERVICE                0x40          /* not
supported */

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 11 and Later Syntax

```

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

/*
 * attMonitorFilterExt() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorFilterExt(
    ATTPrivateData_t      *privateData,
    ATTPrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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
#define ATT_ENDPOINT_UNREGISTERED_FILTER    0x20
#define ATT_ENDPOINT_REGISTERED_FILTER      0x10

/* 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 5-10 Syntax

```

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

/*
 * attMonitorFilterExt() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorFilterExt(
    ATTPrivateData_t      *privateData,
    ATTPrivateFilter_t    privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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
 * Formatting Function
 */

RetCode_t attMonitorFilter(
    ATTPrivateData_t      *privateData,
    ATTV4PrivateFilter_t  privateFilter);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      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:

TSAPI uses the Monitor Ended Event Report to indicate that it will no longer provide events for a monitor previously established through a `cstaMonitorCall()`, `cstaMonitorDevice()` or `cstaMonitorCallsViaDevice()` service request. This may occur because the monitored object has been removed or changed through switch administration to make it invalid, 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:

<code>monitorCrossRefID</code>	[mandatory] Must be a valid Cross Reference ID of this <code>acsOpenStream</code> session.
<code>cause</code>	[optional — supported] Specifies the reason for this event. The following Event Causes are explicitly sent from the switch: • <code>EC_NETWORK_NOT_OBTAINABLE</code> The previously monitored object is no longer available due to a CTI link failure. • <code>EC_RESOURCES_NOT_AVAILABLE</code> The previously monitored object is no longer available or valid due to switch administration changes or because of a communication protocol error.

Detailed Information:

See [Event Report Detailed Information](#) on page 877.

Syntax

```

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

/* CSTAMonitorEndedEvent */

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;
} 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: `CSTAEscapeSvcConfEvent`
- 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 an active call monitor 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 ID that was established by the original `cstaMonitorCall()` service request remains active.

The call monitor will receive 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:

<code>monitorCrossRefID</code>	[mandatory] Must be a valid Cross Reference ID that was returned in a previous <code>CSTAMonitorConfEvent</code> of this <code>acsOpenStream</code> session.
<code>callID</code>	[mandatory] This is the <code>callID</code> of the call whose event reports are to be stopped.

Ack Parameters:

None for this service.

Ack Private Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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 877.

- This service will take effect immediately. Event reports to the application for the specified call will cease after this service 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 monitors.

Syntax

```

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

/* cstaEscapeService() - Service Request */

RetCode_t    cstaEscapeService(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTAEscapeSvcConfEvent - 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 Syntax

```

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

/*
 * attMonitorStopOnCall() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attMonitorStopOnCall(
    ATTPrivateData_t      *privateData,
    CSTAMonitorCrossRefID_t monitorCrossRefID,
    ConnectionID_t        *call);

/* ATTMonitorStopOnCallConfEvent- Service Response Private Data */

typedef struct ATTEvent_t
{
    ATTEventType_t      eventType;          /*
ATT_MONITOR_STOP_ON_CALL_CONF */
    union
    {
        ATTMonitorStopOnCallConfEvent_t monitorStopOnCall;
    } 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.

Service Parameters:

`monitorCrossRefID` [mandatory] Must be a valid Cross Reference ID that was returned in a previous `CSTAMonitorConfEvent` of this `acsOpenStream` session.

Ack Parameters:

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 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `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 [Event Report Detailed Information](#) on page 877.

- **Switch Operation** — This service will take effect immediately. Event reports to the application for calls in progress will stop for the specified monitor. 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;
} CSTAEvent_t;

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


Chapter 11: Event Report Service Group

The *Event Report Service Group* provides event messages (or reports) from Avaya Communication Manager to the Application Enablement Services (AE Services) TSAPI Service.

- [CSTAEventCause and LocalConnectionState](#) on page 625
- [Call Cleared Event](#) on page 633
- [Charge Advice Event \(Private\)](#) on page 639
- [Conferenced Event](#) on page 644
- [Connection Cleared Event](#) on page 668
- [Delivered Event](#) on page 678
- [Diverted Event](#) on page 725
- [Do Not Disturb Event](#) on page 735
- [Endpoint Registered Event \(Private Data Version 11 and later\)](#) on page 737
- [Endpoint Unregistered Event \(Private Data Version 11 and later\)](#) on page 744
- [Entered Digits Event \(Private\)](#) on page 751
- [Established Event](#) on page 754
- [Failed Event](#) on page 789
- [Forwarding Event](#) on page 799
- [Held Event](#) on page 802
- [Logged Off Event](#) on page 807
- [Logged On Event](#) on page 810
- [Private Data Version 14](#)

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>

/* ATTV14LoggedOnEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_LOGGED_ON */
    union
    {
        ATTV14LoggedOnEvent_t    loggedOnEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV14LoggedOnEvent_t {
```

```
        ATTWorkMode_t          workMode;  
    } ATTLeggedOnEvent_t;  
  
typedef enum ATTWorkMode_t {  
    WM_NONE = -1,  
    WM_AUX_WORK = 1,  
    WM_AFTCAL_WK = 2,  
    WM_AUTO_IN = 3,  
    WM_MANUAL_IN = 4  
} ATTWorkMode_t;
```

- Message Waiting Event on page 814
- [Network Reached Event](#) on page 818
- [Originated Event](#) on page 827
- [Queued Event](#) on page 838
- [Retrieved Event](#) on page 845
- [Service Initiated Event](#) on page 848
- [Transferred Event](#) on page 854
- [Event Report Detailed Information](#) on page 877

CSTAEventCause and LocalConnectionState

The Event Report Service Group members described in this chapter rely extensively on the `CSTAEventCause` definitions and `LocalConnectionState` enumerated types.

The following figure provides the definition of the `CSTAEventCause` enumerated type

```
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_NETWORK_SIGNAL = 46,
    EC_SINGLE_STEP_TRANSFER = 52,
    EC_ALERT_TIME_EXPIRED = 60,
    EC_DEST_OUT_OF_ORDER = 65,
    EC_NOT_SUPPORTED_BEARER_SERVICE = 80,
    EC_UNASSIGNED_NUMBER = 81,
    EC_INCOMPATIBLE_BEARER_SERVICE = 87
} CSTAEventCause_t;
```

The following figure provides the definition of the `LocalConnectionState` enumerated type:

```
typedef enum LocalConnectionState_t {
    CS_NONE = -1,                /* state is unknown */
    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 event cause definitions. See [Table 17](#) for a description the cause codes that are possible for each of the call events.

Table 16: Event Cause Definitions

Event Cause	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.

Table 16: Event Cause Definitions

Event Cause	Definition
Call Forward – Immediate	The call has been redirected via a Call Forwarding feature set for all conditions.
Call Forward – Busy	The call has been redirected via a Call Forwarding feature set for a busy endpoint.
Call Forward – 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.
Destination 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 ⁹	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.
Network 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..."
Network Not Obtainable	The call could not reach a destination network.

⁹ 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 16: Event Cause Definitions

Event Cause	Definition
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.
Voice Unit Initiator	Indicates that the event was the result of action by automated equipment (voice mail device, voice response unit, or 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.
Destination 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.

Table 17: CSTA Event Report – Cause Relationships

Cause	Call Clr	Conf	Con. Clr	Div	Div	Est	Fail	Held	Net. Rch	Orig	Q-ed	Retr	Svc Init.	Tran	Cell Call ¹⁰
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								

¹⁰ CTI cause values for cell phones.

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 10
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		
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

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 10
Unassigned Number															y
Incompatible Bearer Service															y

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 receive the Delivered Event.

 NOTE:

The Event Minimization feature must be set to "n" on the switch for the CTI link administered for the Application Enablement Services TSAPI Service.

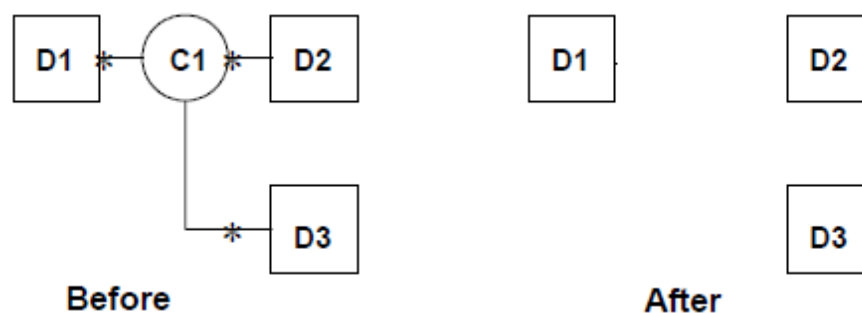
Call Cleared Event

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:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>clearedCall</code>	[mandatory] Specifies the call identifier of the call that has been cleared. The <code>deviceId</code> is set to 0.
<code>localConnectionInfo</code>	[optional – supported] Always specifies a null state (<code>CS_NULL</code>).
<code>cause</code>	[optional – supported] Specifies a cause when the call is not terminated normally. <code>EC_NONE</code> is specified for normal call termination. • <code>EC_BUSY</code> – Device busy. • <code>EC_CALL_CANCELLED</code> – Call rejected or canceled. • <code>EC_DEST_NOT_OBTAINABLE</code> – Called device is not reachable or wrong number is called. • <code>EC_CALL_NOT_ANSWERED</code> – Called device not responding or call not answered (<code>maxRings</code> timed out) for a <code>cstaMakePredictiveCall()</code> service request. • <code>EC_NETWORK_CONGESTION</code> – Network congestion or channel is unacceptable. • <code>EC_RESOURCES_NOT_AVAILABLE</code> – No circuit or channel is available. • <code>EC_SINGLE_STEP_TRANSFER</code> (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 <code>cstaMonitorCallsViaDevice()</code>, but not for a monitor created via <code>cstaMonitorDevice()</code>.) • <code>EC_TRANSFER</code> – Call merged due to transfer or conference. • <code>EC_REORDER_TONE</code> – Intercept SIT treatment – Number changed. • <code>EC_VOICE_UNIT_INITIATOR</code> – Answer machine is detected for a <code>cstaMakePredictiveCall()</code> request.

Private Parameters:`reason`

[optional] Specifies the reason for this event. The following reason codes are supported:

- `AR_NONE` – indicates 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](#) on page 877.

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_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefId;
            union
            {
                CSTACallClearedEvent_t  callCleared;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct CSTACallClearedEvent_t {
    ConnectionID_t      clearedCall;

    /* deviceID = "0" */

    /* devIDType = DYNAMIC_ID */
    LocalConnectionState_t  localConnectionInfo;

    /* always CS_NULL */
    CSTAEventCause_t        cause;
} CSTACallClearedEvent;

```

Private Data 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_t    eventType;          /* ATT_CALL_CLEARED */
    union
    {
        ATTCallClearedEvent_t    callClearedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTCallClearedEvent_t {
    ATTRReasonCode_t    reason;
} ATTCallClearedEvent_t;

typedef enum ATTRReasonCode_t {
    AR_NONE = 0,                                /* no
reason code provided */
    AR_ANSWER_NORMAL = 1,                        /* answer supervision
from the
        * network or internal answer */
    AR_ANSWER_TIMED = 2,                        /* answer
assumed based on
        * internal timer */
    AR_ANSWER_VOICE_ENERGY = 3,                /* voice energy detection by
call
        * 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 */
} ATTRReasonCode_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 monitor, a monitor of all trunk groups, a station monitor, or a call monitor. This event is available only if trunk group monitoring (or monitoring of all trunk groups) is requested to the switch for turning the Charge Advice feature on.

Service Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.

Private Parameters:

<code>connection</code>	[mandatory] Specifies the <code>connectionID</code> of the trunk party that generated the charge event. The <code>deviceID</code> is null if split charge is reported due to a conference or transfer.
<code>calledDevice</code>	[mandatory] Specifies the external device that was dialed or requested. This number does not include ARS, FAC, or TAC digits.
<code>chargingDevice</code>	[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.
<code>trunkGroup</code>	[mandatory] Specifies the trunk group receiving the charge. The number provided corresponds to the number used in switch administration, and is not the Trunk Access Code.
<code>trunkMember</code>	[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 – the network failed to provide a final charge for the call (CS3/38)
- CE_LESS_FINAL_CHARGE – the final charge provided by the network is less than a previous charge (CS3/38)
- CE_CHARGE_TOO_LARGE – the 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 `attSetAdviceOfCharge()`.
- Trunk Group Administration – Only ISDN-PRI trunk groups that have Charge Advice set to "during-on-request" or "automatic" on the switch will provide 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_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_PRIVATE_STATUS */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    struct
    {
        CSTAMonitorCrossRefID_t monitorCrossRefId;
        union
        {
            CSTAPrivateStatusEvent_t privateStatus;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct    CSTAPrivateStatusEvent_t {
    Nulltype      null;
} CSTAPrivateStatusEvent_t;

```


Private Data 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;
```

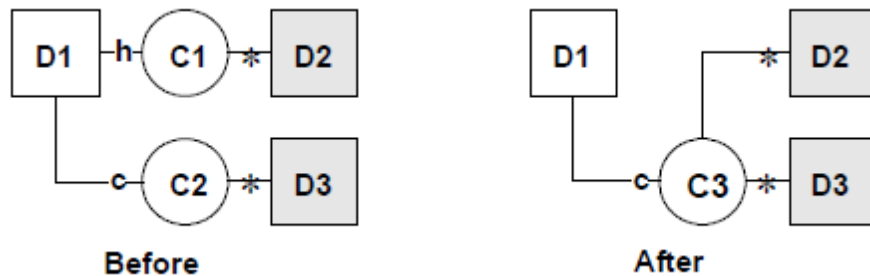
Conferenced Event

Summary

- **Direction:** Switch to Client
- **Event:** CSTAConferencedEvent
- **Private Data Event:** ATTConferencedEvent (private data version 7 and later), 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 a TSAPI application successfully completes a `cstaConferenceCall()` request.
- When a TSAPI application successfully completes an `attSingleStep-ConferenceCall()` request.
- When the "call park" feature is used in conjunction with the "conference" button on the voice set.

Service Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>primaryOldCall</code>	[mandatory] Specifies the <code>callID</code> 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.
<code>secondaryOldCall</code>	[mandatory] Specifies the <code>callID</code> 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.
<code>confController</code>	[mandatory] Specifies the device that is controlling the conference. This is the device that set up the conference.
<code>addedParty</code>	[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` has status `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.

<code>conferenceConnections</code>	[optional – supported] Specifies a count of the number of devices and a list of <code>connectionIDs</code> and <code>deviceIDs</code> 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 <code>deviceID</code> 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.

<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for the <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> indicates that the local connection state is unknown.

cause

[optional – limited support] 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 participation type PT_ACTIVE. For details, see [Single Step Conference Call Service \(Private Data Version 5 and Later\)](#) on page 410 in Chapter 6.

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 410 in Chapter 6.

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.
- `trunkMember` – the original `trunkMember` received by the `heldCall` or the `primaryOldCall`. This parameter is supported by private data version 5 and later.
- `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.
- For private data versions 2-5, the maximum length of `userInfo` is 32 bytes. Beginning with private data version 6, 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 bytes of data for `userInfo`, regardless of the size of the data 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` – 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 650.
 - `olddeviceID`
 - `cause`
 - `oldconnectionID`

This parameter is supported by private data version 7 and later.

`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.

<code>distributingVDN</code>	The VDN extension associated with the distributing device. This field gets set only and exactly under the following conditions. • When the application monitors the VDN in question and a call is offered to the VDN. (This appears to the VDN monitor as 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).
<code>ucid</code>	[optional] Specifies the Universal Call ID (UCID) of the resulting <code>newCall</code>. 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 <code>ucid</code> contains the <code>ATT_NULL_UCID</code> (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.
<code>trunkList</code>	[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 parameters are supported: • <code>count</code> – The count of the connected parties on the call. • <code>trunks</code> – An array of 5 trunk group and trunk member IDs, one for each connected party. The following parameters are supported: – <code>connection</code> – The connection ID of one of the parties on the call. – <code>trunkGroup</code> – The trunk group of the party referenced by <code>connection</code>. – <code>trunkMember</code> – The trunk member of the party referenced by <code>connection</code>.

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).

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `DeviceID` that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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_NETWORK_SIGNAL` 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:

See also the [Event Report Detailed Information](#) on page 877.

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` will 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 that ended as the result of the `cstaConferenceCall()` request) 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 `distributingDevice` will be sent in the Conferenced Event. The `primaryOldCall` (the call that 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_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_CONFERENCED */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefId;
            union
            {
                CSTAConferencedEvent_t  conferenced;
            } u;
        } cstaUnsolicited;
    } event;
} 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 {
    unsigned int        count;
    Connection_t        *connection;
} ConnectionList_t;

```

Private Data 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.

Private Data Version 7 and Later Syntax

The `deviceHistory` and `distributingVDN` parameters are new for private data version 7.

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

/* ATTConferencedEvent - CSTA Unsolicited Event Private Data */

typedef struct ATTEvent_t
{
    ATTEventType_t    eventType;          /* ATT_CONFERENCED */
    union
    {
        ATTConferencedEvent_t    conferencedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTConferencedEvent_t {
    ATTOriginalCallInfo_t    originalCallInfo;
    CalledDeviceID_t          distributingDevice;
    ATTUCID_t                 ucid;
    ATTTTrunkList_t           trunkList;
    DeviceHistory_t           deviceHistory;
    CalledDeviceID_t          distributingVDN;
} 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;
    unsigned char              flexibleBilling;
    DeviceHistory_t           deviceHistory;
} ATTOriginalCallInfo_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_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 struct ATTUnicodeDeviceID_t {
    unsigned short               count;
    short                        value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE      25

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 provided */
    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;

#define ATT_MAX_USER_INFO    129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t    type;
    struct
    {
        short                length;        /* 0
indicates no UUI */
        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 */
} ATTUUIProtocolType_t;

typedef char    ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char    hasInfo;        /* if FALSE, no
call originator info */
    short            callOriginatorType;
} ATTCallOriginatorInfo_t;

typedef struct DeviceHistory_t {
    unsigned int        count;        /* at most 1 */
    DeviceHistoryEntry_t    *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t            olddeviceID;
    CSTAEventCause_t        cause;
    ConnectionID_t        oldconnectionID;
} DeviceHistoryEntry_t;

```

Private Data Version 6 Syntax

```

#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 ATTV6ConferencedEvent_t {
    ATTV6OriginalCallInfo_t    originalCallInfo;
    CalledDeviceID_t            distributingDevice;
    ATTUCID_t                    ucid;
    ATTTTrunkList_t              trunkList;
} ATTV6ConferencedEvent_t;

typedef struct ATTV6OriginalCallInfo_t {
    ATTRReasonForCallInfo_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;
    unsigned char                flexibleBilling;
} ATTV6OriginalCallInfo_t;

typedef enum ATTRReasonForCallInfo_t {
    OR_NONE = 0,                /* indicates not present */
    OR_CONSULTATION = 1,
    OR_CONFERENCED = 2,
    OR_TRANSFERRED = 3,
    OR_NEW_CALL = 4
} ATTRReasonForCallInfo_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 struct ATTUnicodeDeviceID_t {
    unsigned short    count;
    short             value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE    25

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 provided */
    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;

#define ATT_MAX_USER_INFO      129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;      /* 0
indicates no UUI */
        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 {
    unsigned char  hasInfo;        /* if FALSE, no
                                     *
call originator info */
    short          callOriginatorType;
} ATTCallOriginatorInfo_t;

```


Private Data Version 5 Syntax

```

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

/* ATTV5ConferencedEvent - CSTA Unsolicited Event Private Data */

typedef struct ATTEvent_t
{
    ATTEventType_t    eventType;          /* ATTV5_CONFERENCED */
    union
    {
        ATTV5ConferencedEvent_t    v5conferencedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV5ConferencedEvent_t
{
    ATTV5OriginalCallInfo_t    originalCallInfo;
    CalledDeviceID_t            distributingDevice;
    ATTUCID_t                    ucid;
} ATTV5ConferencedEvent_t;

typedef struct ATTV5OriginalCallInfo_t {
    ATTRReasonForCallInfo_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;
    unsigned char                flexibleBilling;
} ATTV5OriginalCallInfo_t;

typedef enum ATTRReasonForCallInfo_t {
    OR_NONE = 0,                /* indicates not present */
    OR_CONSULTATION = 1,
    OR_CONFERENCED = 2,
    OR_TRANSFERRED = 3,
    OR_NEW_CALL = 4
} ATTRReasonForCallInfo_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 struct ATTUnicodeDeviceID_t {
    unsigned short    count;
    short             value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE    25

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 provided */
    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 no UUI */
        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 */
} ATTUUIProtocolType_t;

typedef char    ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char    hasInfo;           /* if FALSE, no
call originator info */
    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_t          eventType;          /* ATTV4_CONFERENCED */
    union
    {
        ATTV4ConferencedEvent_t      v4conferencedEvent;
    } 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_t      CallingDeviceID_t;
typedef ExtendedDeviceID_t      CalledDeviceID_t;

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 enum ATTUserEnteredCodeType_t {
    UE_NONE = -1,                        /* indicates not provided */
    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 no UUI */
        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 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_t    eventType;          /* ATTV3_CONFERENCED */
    union
    {
        ATTV3ConferencedEvent_t    v3conferencedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV3ConferencedEvent_t
{
    ATTV4OriginalCallInfo_t    originalCallInfo;
} ATTV3ConferencedEvent_t;

typedef struct ATTV4OriginalCallInfo_t {
    ATTRReasonForCallInfo_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 ATTRReasonForCallInfo_t {
    OR_NONE = 0,                /* indicates not present */
    OR_CONSULTATION = 1,
    OR_CONFERENCED = 2,
    OR_TRANSFERRED = 3,
    OR_NEW_CALL = 4
} ATTRReasonForCallInfo_t;

typedef ExtendedDeviceID_t      CallingDeviceID_t;
typedef ExtendedDeviceID_t      CalledDeviceID_t;

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 enum ATTUserEnteredCodeType_t {
    UE_NONE = -1,                   /* indicates not provided */
    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 no UUI */
        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;
```

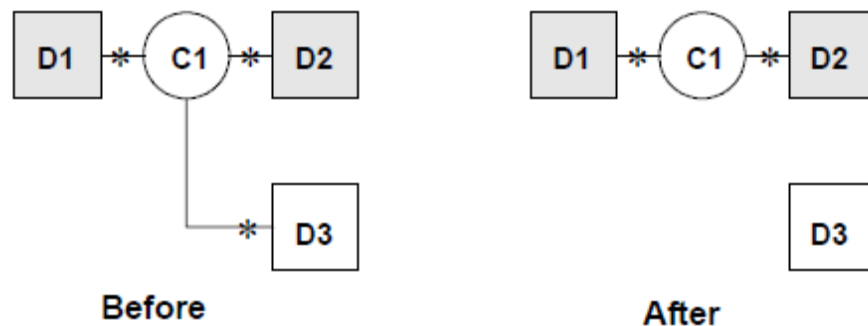
Connection Cleared Event

Summary

- Direction: Switch to Client
- Event: CSTAConnectionClearedEvent
- Private Data Event: ATTConnectionClearedEvent (private data version 7 and later), ATTV6ConnectionClearedEvent (private data version 6), ATTV5ConnectionClearedEvent (private data versions 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 predictive 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 that case, a Call Cleared Event Report is generated instead.

This event is the last event of a call for a `cstaMonitorDevice()` request.

Service Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>droppedConnection</code>	[mandatory] Specifies the connection that has been dropped from the call.
<code>releasingDevice</code>	[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.
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> 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 `cstaMakePredictiveCall()` request.
- `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 `cstaMakePredictiveCall()` request.
- `EC_INCOMPATIBLE_BEARER_SERVICE` – The connection was cleared because the selected facility for the call did not have the proper bearer capability. This could occur, for example, if a data or video call was attempted on a trunk facility that is reserved for voice traffic.
- `EC_REDIRECTED` – A queued connection has been redirected to a station.

Private Parameters:`userInfo`

[optional] Contains user-to-user information. This parameter allows an application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) 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()` service request with `userInfo` and passed to an application in the Connection Cleared Event Report.

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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).

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `DeviceID` that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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_NETWORK_SIGNAL` 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:

See the [Event Report Detailed Information](#) on page 877.

Syntax

```

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

/* CSTAConnectionClearedEvent */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;      /* CSTAUNSOLICITED */
    EventType_t      eventType;      /* CSTA_CONNECTION_CLEARED */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefId;
            union
            {
                CSTAConnectionClearedEvent_t
            }
            connectionCleared;
        } u;
    } cstaUnsolicited;
} event;
} CSTAEvent_t;

typedef struct CSTAConnectionClearedEvent_t {
    ConnectionID_t      droppedConnection;
    SubjectDeviceID_t   releasingDevice;
    LocalConnectionState_t  localConnectionInfo;
    CSTAEventCause_t    cause;
} CSTAConnectionClearedEvent_t;

```

Private Data 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.

Private Data Version 7 and Later Syntax

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
{
    ATTEventType_t    eventType;          /* ATT_CONNECTION_CLEARED */
    union
    {
        ATTConnectionClearedEvent_t    connectionCleared;
    } u;
} ATTEvent_t;

typedef struct ATTConnectionClearedEvent_t {
    ATTUserToUserInfo_t    userInfo;
    DeviceHistory_t        deviceHistory;
} ATTConnectionClearedEvent_t;

#define ATT_MAX_USER_INFO    129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t    type;
    struct
    {
        short                length;      /* 0
indicates no UUI */
        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

```

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

/* ATTV6ConnectionClearedEvent - CSTA Unsolicited Event Private Data
*/

typedef struct
{
    ATTEventType_t      eventType;          /*
ATTV6_CONNECTION_CLEARED */
    union
    {
        ATTV6ConnectionClearedEvent_t      v6connectionCleared;
    } u;
} ATTEvent_t;

typedef struct ATTV6ConnectionClearedEvent_t {
    ATTUserToUserInfo_t      userInfo;
} ATTV6ConnectionClearedEvent_t;

#define ATT_MAX_USER_INFO      129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;      /* 0
indicates no UUI */
        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>

/* ATTV5ConnectionClearedEvent - CSTA Unsolicited Event Private Data
*/

typedef struct
{
    ATTEventType_t      eventType;          /*
ATTV5_CONNECTION_CLEARED */
    union
    {
        ATTV5ConnectionClearedEvent_t      v5connectionCleared;
    } u;
} ATTEvent_t;

typedef struct ATTV5ConnectionClearedEvent_t {
    ATTV5UserToUserInfo_t      userInfo;
} ATTV5ConnectionClearedEvent_t;

typedef struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;      /* 0
indicates no UUI */
        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 13 and later), ATTV12DeliveredEvent (private data version 12), 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, distributingVDN, channelType

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 Communication Manager devices that distribute calls, VDNs and ACD splits.

A Delivered Event Report is generated when a call is delivered to a monitored VDN or ACD split:

- Call Delivered to a VDN – An event is generated when a call is delivered to a monitored VDN.
- Call Delivered to an ACD Split – An event is generated when a call is delivered to a monitored ACD split. The event report will be sent even if the ACD split is in night service or has call forwarding active.

A Delivered Event 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:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>connection</code>	[mandatory] Specifies the endpoint that is alerting.
<code>alertingDevice</code>	[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.
<code>callingDevice</code>	[mandatory] Specifies the calling device. The following rules apply: • For internal calls – the originator's extension. • For outgoing calls over PRI facilities – the "calling number" from the ISDN SETUP message is specified. If the "calling number" is not available, the assigned trunk identifier is provided instead. 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 – the "calling number" from the ISDN SETUP message is specified. If the "calling number" is not available, the assigned trunk identifier is provided instead.

- For incoming calls over non-PRI facilities – the calling party number is generally not available. The assigned trunk identifier is provided instead.
- The trunk identifier is specified only when the calling party number is not available.

**NOTE:**

The trunk identifier is a dynamic device identifier. It cannot be used to access a trunk in Communication Manager.

- For calls originated at a bridged call appearance – the principal's extension is specified.
- There is a special case for a predictive 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 an invalid number and the Invalid Number Dialed Intercept Treatment type on Avaya Communication Manager is administered as “announcement” (rather than “tone”), then the extension number of the announcement 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 causes EC_CALL_FORWARD, EC_CALL_FORWARD_ALWAYS, EC_CALL_FORWARD_BUSY, and EC_CALL_FORWARD_NO_ANSWER have higher precedence than causes: EC_KEY_CONFERENCE, EC_NEW_CALL, and EC_REDIRECTED.

When more than one cause applies to an event, the cause with the highest precedence is reported. For example, if causes EC_CALL_FORWARD_ALWAYS and EC_KEY_CONFERENCE both apply, the event cause reported in the event will be EC_CALL_FORWARD_ALWAYS because it has higher precedence.

- EC_CALL_FORWARD – 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 – The call has been redirected via the Call Forwarding feature.
- EC_CALL_FORWARD_BUSY – The call has been redirected for one of the following reasons:
 - Cover – principal busy
 - Cover – all call appearance busy

- `EC_CALL_FORWARD_NO_ANSWER` – The call has received Call Coverage treatment because the monitored station did not answer the call within the administered number of rings.
- `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.
- `EC_TRANSFER` (private data versions 2-7) – The call has been delivered to the alerting device as the result of a Single Step Transfer Call Operation.
- `EC_SINGLE_STEP_TRANSFER` (private data versions 8 or later) – The call has been delivered to the alerting device as the result of a Single Step Transfer Call Operation.

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.

The `deliveredType` is `DELIVERED_TO_ACD` when either:

- The application uses `cstaMonitorCallsViaDevice()` to monitor an ACD split or VDN, and a call is offered to the monitored device.
- A converse-on vector step delivers a call to an agent or automated attendant while maintaining the call's position in other queues.

For this `deliveredType`, the `split` and `distributingDevice` parameters are normally not set. The `distributingVDN` parameter provides the ACD split or VDN that delivered the call (i.e., the device ID of the monitored device), irrespective of the administration of the VDN Override feature.

- `DELIVERED_TO_STATION` – This type indicates that the call is delivered to a station.

For this `deliveredType`:

- If the call has been delivered to an ACD agent, then the `split` parameter provides the extension number of the ACD split that delivered the call.
- If the application is using `cstaMonitorCallsViaDevice()` to monitor the ACD split or VDN that delivered the call, then the `distributingDevice` parameter provides the ACD split or VDN that delivered the call (subject to the administration of the VDN Override feature on Avaya CM), and the `distributingVDN` parameter provides the ACD split or VDN that delivered the call (i.e., the device ID of the monitored device), irrespective of the administration of the VDN Override feature.

<code>trunkGroup</code>	[optional] Specifies the trunk group number from which the call originated. This parameter is supported by private data version 5 and later only.
<code>trunk</code>	[optional] Specifies the trunk group number from which the call originated. Trunk group number is provided only if the <code>callingDevice</code> is unavailable. This parameter is supported by private data versions 2, 3, and 4 only.
<code>trunkMember</code>	[optional – limited support] Specifies the trunk member number from which the call originated. This parameter is supported by private data version 5 and later.
<code>split</code>	[optional] Specifies the ACD split extension which delivered the call to the agent. This parameter applies to <code>deliveredType</code> <code>DELIVERED_TO_STATION</code> only.
<code>lookaheadInfo</code>	[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.
<code>userEnteredCode</code>	[optional] Specifies the code/digits that may have been entered by the caller through the Communication Manager call prompting feature or the collected digits feature. If the <code>userEnteredCode</code> code is set to "UE_NONE", no <code>userEnteredCode</code> private data is provided with this event. See the Detailed Information section for how to setup the switch and application for collecting <code>userEnteredCode</code> .

<code>userInfo</code>	[optional] Contains user-to-user information. This parameter allows an application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) bytes, with a call. This information may be a customer number, credit card number, alphanumeric digits, or a binary string. An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in <code>userInfo</code>, regardless of the size of the data sent by the switch. The following <code>UII</code> protocol types are supported: <code>UII_NONE</code> – There is no data provided in the data parameter. <code>UII_USER_SPECIFIC</code> – The content of the <code>data</code> parameter is a binary string. The correct size (maximum of 32 or 96 bytes) of data must be specified in the <code>size</code> parameter. <code>UII_IA5_ASCII</code> – The content of the <code>data</code> 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 <code>size</code> parameter.
<code>reason</code>	[optional] Specifies the reason of this event. The following reasons are supported: • <code>AR_NONE</code> – indicates no value specified for reason. • <code>AR_IN_QUEUE</code> – 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 <code>DELIVERED_TO_ACD</code> only. Otherwise, this parameter will be set to <code>AR_NONE</code>.
<code>originalCallInfo</code>	[optional] Specifies the original call information. Note that information is not repeated in the <code>originalCallInfo</code>, if it is already reported in the CSTA service parameters or in the private data. For example, the <code>callingDevice</code> and <code>calledDevice</code> in the <code>originalCallInfo</code> will be <code>NULL</code> if the <code>callingDevice</code> and the <code>calledDevice</code> in the CSTA service parameters are the original calling and called devices. Only when the original devices are different from the most recent <code>callingDevice</code> and <code>calledDevice</code>, the <code>callingDevice</code> and <code>calledDevice</code> in the <code>originalCallInfo</code> will be set. If the <code>userEnteredCode</code> in the private data is the original <code>userEnteredCode</code>, the <code>userEnteredCode</code> in the <code>originalCallInfo</code> will be <code>UE_NONE</code>. Only when new <code>userEnteredCode</code> is received and reported in the <code>userEnteredCode</code>, the <code>originalCallInfo</code> will have the original <code>userEnteredCode</code>.

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.
- `trunkGroup` – the original `trunkGroup` received by the `activeCall`. This parameter is supported by private data version 5 and later only.
- `trunkMember` – 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`.

Note: An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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 689.
 - `olddeviceId`
 - `cause`
 - `oldconnectionID`

This parameter is supported by private data version 7 and later.

`distributingDevice`

[optional] Specifies the ACD or VDN device that distributed the call to the agent station. This information is only provided when the application is using `cstaMonitorCallsViaDevice()` to monitor the ACD split or VDN that delivered the call to the station, and is provided for station monitors only (that is, when the `deliveredType` is `DELIVERED_TO_STATION`). This parameter is supported by private data version 4 and later.

The value of the `distributingDevice` may be affected by CM administration of the VDN Override feature. The `calledDevice` specifies the originally called device. In many ACD call scenarios, the `calledDevice` and the `distributingDevice` have the same device ID. However, in call scenarios that involve call vectoring with the VDN Override feature turned on, the `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 involve more than one VDN. If the VDN Override feature is enabled for the `calledDevice`, then the `distributingDevice` specifies the VDN that distributed the call to the agent. This is particularly useful for applications that need to know the call path.

For example, suppose that the VDN Override feature is enabled for VDN 25201. (That is, the “Allow VDN Override” field on the vector directory number (vdn) form is set to “y”. Also, if the VDN will receive incoming trunk calls, then the “VDN Override for ASAI Messages” field is set to either “ISDN Trunk” or “all”, as appropriate.)

VDN 25201 can route the call to either VDN 25202 or VDN 25203. The VDN Override feature is not enabled on either of these VDNs. Both of these VDNs route the call to VDN 25204. Then VDN 25204 routes the call to an agent.

If VDN 25201 and the agent station are both monitored, but VDN 25202 and VDN 25203 are not monitored, then for a call to VDN 25201, the `distributingDevice` in the Delivered and Established events received by the agent station monitor will indicate whether the call path included VDN 24202 or VDN 24203.

Also, within the Delivered and Established events received by the agent station monitor, the `calledDevice` and the `lastRedirectionDevice` will be 25201. (But if VDN 25204 is also monitored, then the `lastRedirectionDevice` would be 25204).

Note: Proper switch administration of the VDN Override feature is required on Avaya 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. Unlike the `distributingDevice` field, the value of the `distributingVDN` field is not affected by CM administration of the VDN Override feature. The field gets set only and exactly under the following conditions.

- When the application monitors the VDN in question and a call is offered to the VDN. This event is conveyed to the applications as 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.

<code>callOriginatorInfo</code>	[optional] Specifies the <code>callOriginatorType</code> of the call originator such as coin call, 800-service call, or cellular call. See Table 18 on page 696. Note: <code>callOriginatorType</code> values (II digit assignments) are provided by the network, not 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
<code>flexibleBilling</code>	[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 <code>TRUE</code>, 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.
<code>deviceHistory</code>	The <code>deviceHistory</code> parameter type specifies a list of <code>deviceIDs</code> 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 <code>deviceHistory</code> 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). Conceptually, the <code>deviceHistory</code> parameter consists of a list of entries, where each entry contains information about a <code>deviceID</code> that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry. The entry consists of: • <code>olddeviceID</code> – 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 <code>divertingDevice</code> 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 may also be: – “Not Known” – indicates that the device identifier cannot be provided. – “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and <code>oldconnectionID</code> are not provided.

- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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_NETWORK_SIGNAL` 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 below, see the [Event Report Detailed Information](#) on page 877.

- 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 illustrate how this parameter may be set and some of 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 `lastRedirectionDevice` in the Delivered/Established Events sent to both the caller and the agent will have a `deviceIDStatus` of `ID_NOT_REQUIRED`.
 - If the caller calls the VDN, instead of the ACD Split, and the VDN sends the call to the ACD Split, the `lastRedirectionDevice` in the Delivered/Established Events sent to both the caller and the agent will have a `deviceIDStatus` of `ID_NOT_REQUIRED`. The last redirection device in the PBX is actually the ACD Split.
 - 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 ACD Split as the `last-RedirectionDevice`.
 - 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.
 - If the caller dials the ACD Split directly, the Delivered/Established Events sent to both the caller and the agent will have the VDN as the `lastRedirection-Device`.
 - 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.
 - 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 `last-RedirectionDevice`.
 - 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 the 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.
 - The `lastRedirectionDevice` in the Delivered/Established Events sent to the answering party will have a `deviceIDStatus` of `ID_NOT_REQUIRED`.

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 User Entered Codes (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.

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](#).

1. 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:

Suppose VDN 24101 is mapped to vector 1, and vector 1 has the following steps:

1. Collect 16 digits after announcement extension 1000
2. Route to 24102
3. Stop

Suppose VDN 24102 is mapped to vector 2, and 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:

Suppose VDN 24201 is mapped to vector 11, and vector 11 has the following steps:

1. Collect 10 digits after announcement extension 2000.
2. Route to 24202.
3. Stop.

Suppose VDN 24202 is mapped to vector 12, and vector 12 has the following steps:

1. Collect 16 digits after announcement extension 3000.
2. Route to 24203.
3. Stop.

Suppose VDN 24203 is mapped to vector 13, and 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.
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 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

Table 18: Call Originator Type Values (II-digits)

Code	Description
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
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.

Table 18: Call Originator Type Values (II-digits)

Code	Description
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 pay stations 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.
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

Table 18: Call Originator Type Values (II-digits)

Code	Description
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.)
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

Table 18: Call Originator Type Values (II-digits)

Code	Description
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

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_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t      monitorCrossRefId;
            union
            {
                CSTADeliveredEvent_t      delivered;
            } u;
        } cstaUnsolicited;
    } event;
} 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 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.

Private Data Version 13 and Later Syntax

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
{
    ATTEventType_t    eventType;          /* ATT_DELIVERED */
    union
    {
        ATTDeliveredEvent_t    deliveredEvent;
    } 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;
    ATTCallOriginatorInfo_t    callOriginatorInfo;
    unsigned char          flexibleBilling;
    DeviceHistory_t        deviceHistory;
    CalledDeviceID_t        distributingVDN;
    ATTChannelType_t        channelType;
} ATTDeliveredEvent_t;

typedef enum ATTChannelType_t {
    CH_OTHER = -1,
    CH_UNKNOWN = 0,
```

```

        CH_VOICE = 1,
        CH_VIDEO = 2
    } ATTChannelType_t;

typedef enum ATTDeliveredType_t {
    DELIVERED_TO_ACD = 1,
    DELIVERED_TO_STATION = 2,
    DELIVERED_OTHER = 3                /* not in use */
} ATTDeliveredType_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 struct ATTUnicodeDeviceID_t {
    unsigned short                count;
    short                        value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE        25

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 provided */
    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;

#define ATT_MAX_USER_INFO      129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;        /* 0
indicates no UUI */
        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 */
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t {
    AR_NONE = 0,                  /* no
reason code provided */
    AR_ANSWER_NORMAL = 1,         /* answer supervision
from the
    * network or internal answer */
    AR_ANSWER_TIMED = 2,          /* answer
assumed based on

```

```

    * internal timer */
    AR_ANSWER_VOICE_ENERGY = 3,          /* voice energy detection by
call

    * 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 */
} ATReasonCode_t

typedef struct ATOriginalCallInfo_t {
    ATReasonForCallInfo_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;
    unsigned char               flexibleBilling;
    DeviceHistory_t             deviceHistory;
} ATOriginalCallInfo_t;

typedef enum ATReasonForCallInfo_t {
    OR_NONE = 0,                  /* indicates not present */
    OR_CONSULTATION = 1,
    OR_CONFERENCED = 2,
    OR_TRANSFERRED = 3,
    OR_NEW_CALL = 4

```

```

} ATReasonForCallInfo_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 struct ATTUnicodeDeviceID_t {
    unsigned short              count;
    short                       value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE      25

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 provided */

```



```

    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;

#define ATT_MAX_USER_INFO      129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;          /* 0
indicates no UUI */
        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 */
} ATTUUIProtocolType_t;

character string */
} ATTUUIProtocolType_t;

typedef char      ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char    hasInfo;          /* if FALSE, no
call originator info */
    short            callOriginatorType;
} ATTCallOriginatorInfo_t;

typedef struct DeviceHistory_t {
    unsigned int      count;          /* at most 1 */
    DeviceHistoryEntry_t *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {

```

```
DeviceID_t           olddeviceID;
CSTAEventCause_t     cause;
ConnectionID_t       oldconnectionID;
} DeviceHistoryEntry_t;
```

Private Data Version 6 Syntax

For private data version 6, the maximum size of the data provided in the `userInfo` parameter is increased from 32 bytes to 96 bytes.

```
#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 ATTV6DeliveredEvent_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;
    ATTV6OriginalCallInfo_t    originalCallInfo;
    CalledDeviceID_t       distributingDevice;
    ATTUCID_t              ucid;
    ATTCallOriginatorType_t    callOriginatorInfo;
    unsigned char          flexibleBilling;
} ATTV6DeliveredEvent_t;

typedef enum ATTDeliveredType_t {
    DELIVERED_TO_ACD = 1,
    DELIVERED_TO_STATION = 2,
    DELIVERED_OTHER = 3                /* not in use */
} ATTDeliveredType_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_VECTURING_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 {
    unsigned short      count;
    short               value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE      25

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 provided */
    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;

```

```

#define ATT_MAX_USER_INFO      129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;          /* 0
indicates no UUI */
        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 */
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t {
    AR_NONE = 0,                  /* no
reason code provided */
    AR_ANSWER_NORMAL = 1,         /* answer supervision
from the
    * network or internal answer */
    AR_ANSWER_TIMED = 2,          /* answer
assumed based on
    * internal timer */
    AR_ANSWER_VOICE_ENERGY = 3,   /* voice energy detection by
call
    * 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

typedef struct ATTV6OriginalCallInfo_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;
    unsigned char              flexibleBilling;
} ATTV6OriginalCallInfo_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_t    CallingDeviceID_t;
typedef ExtendedDeviceID_t    CalledDeviceID_t;

typedef char                  ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char              hasInfo;    /* if FALSE, no
                                           * call originator info */
    short                      callOriginatorType;
} ATTCallOriginatorInfo_t;

```

Private Data Version 5 Syntax

Private data version 5 adds support for the `ucid`, `callOriginatorInfo`, `flexibleBilling`, and `uSourceVDN` parameters.

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

/* ATTV5DeliveredEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t          eventType;          /* ATTV5_DELIVERED */
    union
    {
        ATTV5DeliveredEvent_t    v5deliveredEvent;
    } 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;
    unsigned char               flexibleBilling;
} ATTV5DeliveredEvent_t;

typedef enum ATTDeliveredType_t {
    DELIVERED_TO_ACD = 1,
    DELIVERED_TO_STATION = 2,
    DELIVERED_OTHER = 3                      /* not in use */
} ATTDeliveredType_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_VECTURING_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 {
    unsigned short      count;
    short               value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE      25

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 provided */
    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 no UUI */
        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 */
} ATTUUIProtocolType_t;

typedef enum ATTRReasonCode_t {
    AR_NONE = 0,                  /* no
reason code provided */
    AR_ANSWER_NORMAL = 1,         /* answer supervision
from the
    * network or internal answer */
    AR_ANSWER_TIMED = 2,          /* answer
assumed based on
    * internal timer */
    AR_ANSWER_VOICE_ENERGY = 3,   /* voice energy detection by
call
    * 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 */
    } ATReasonCode_t

typedef struct ATTV5OriginalCallInfo_t {
    ATReasonForCallInfo_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;
    unsigned char            flexibleBilling;
} ATTV5OriginalCallInfo_t;

typedef enum ATReasonForCallInfo_t {
    OR_NONE = 0,                /* indicates not present */
    OR_CONSULTATION = 1,
    OR_CONFERENCED = 2,
    OR_TRANSFERRED = 3,
    OR_NEW_CALL = 4
} ATReasonForCallInfo_t;

typedef ExtendedDeviceID_t    CallingDeviceID_t;
typedef ExtendedDeviceID_t    CalledDeviceID_t;

typedef char                  ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char              hasInfo;    /* if FALSE, no
                                           * call originator info */
    short                      callOriginatorType;
} ATTCallOriginatorInfo_t;

```

Private Data Version 4 Syntax

Private data version 4 adds support for the `distributingDevice` parameter.

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

/* ATTV4DeliveredEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV4_DELIVERED */
    union
    {
        ATTV4DeliveredEvent_t    v4deliveredEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV4DeliveredEvent_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_t    originalCallInfo;
    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 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;

#define ATT_MAX_USER_CODE      25

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 provided */
    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 no UUI */
        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 ATTRReasonCode_t {
        AR_NONE = 0,                                  /* no
reason code provided */
        AR_ANSWER_NORMAL = 1,                          /* answer supervision
from the
        * network or internal answer */
        AR_ANSWER_TIMED = 2,                            /* answer
assumed based on
        * internal timer */
        AR_ANSWER_VOICE_ENERGY = 3,                    /* voice energy detection by
call
        * 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 */
} ATTRReasonCode_t

typedef struct ATTV4OriginalCallInfo_t {
        ATTRReasonForCallInfo_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_t      CallingDeviceID_t;
typedef ExtendedDeviceID_t      CalledDeviceID_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_t    eventType;          /* ATTV3_DELIVERED */
    union
    {
        ATTV3DeliveredEvent_t    v3deliveredEvent;
    } 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_t    originalCallInfo;
} ATTV3DeliveredEvent_t;

typedef enum ATTDeliveredType_t {
    DELIVERED_TO_ACD = 1,
    DELIVERED_TO_STATION = 2,
    DELIVERED_OTHER = 3                /* not in use */
} ATTDeliveredType_t;

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;

#define ATT_MAX_USER_CODE      25

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 provided */
    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 no UUI */
        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 provided */
        AR_ANSWER_NORMAL = 1,                            /* answer supervision
from the
        * network or internal answer */
        AR_ANSWER_TIMED = 2,                             /* answer
assumed based on
        * internal timer */
        AR_ANSWER_VOICE_ENERGY = 3,                     /* voice energy detection by
call
        * 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

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_t           CallingDeviceID_t;
typedef ExtendedDeviceID_t           CalledDeviceID_t;

```

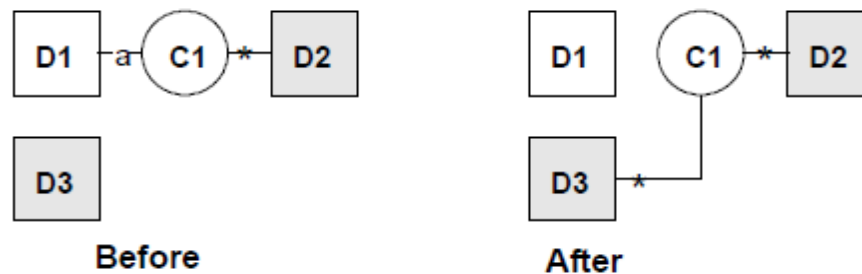
Diverted Event

Summary

- Direction: Switch to Client
- Event: CSTADivertedEvent
- Private Data Event: ATTDivertedEvent (private data version 12 and later), ATTV11DivertedEvent (private data versions 7-11)
- Service Parameters: monitorCrossRefID, connection, divertingDevice, newDestination, localConnectionInfo, cause
- Private Parameter: deviceHistory, callingDevice

Functional Description:

The Diverted Event Report indicates that a call 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 a call has been deflected or diverted from a monitored device. Circumstances for which this event report is sent include:

- When a call enters a new VDN or ACD split that is being monitored.¹¹ For example, if a call leaves one monitored ACD device and enters another, a Diverted Event Report is sent to the monitor for the first ACD device, indicating that the monitor for the first ACD device will no longer receive events for the call.
- When an alerting call leaves a monitored station without having been dropped or disconnected, this report is sent to the station monitor.

Possible reasons why a call that had been alerting at the station may leave the station include:

- 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 voice mail coverage and the Coverage Caller Response Interval (CRI) has elapsed (the principal's call is redirected).

¹¹ 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 call was redirected using the `cstaDeflectCall()` or `cstaPickupCall()` service.
- 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 retains a simulated bridge appearance until the call is dropped/disconnected. Examples of situations when this event is **not** sent are:

- Bridging
- Calls to a TEG (multifunction set with TEG button)
- Incoming PCOL calls (multifunction sets)
- Pick-up for multifunction set principals

For ASAI Link Versions 1-6, this event report is always preceded by a Delivered Event Report for the call arriving at the monitored device, so a Diverted Event is not sent for a call that did not alert at the monitored device.

Beginning with ASAI Link Version 7, station monitors receive a Diverted Event for a call that never alerted at the monitored device in the following scenarios:

- Call Coverage – A call to a monitored station extension receives Call Coverage treatment without first alerting at the station.
- Call Forwarding – A call to a monitored station extension is immediately forwarded without first alerting at the station because one of the following features is active at the station:
 - Call Forwarding – All Calls
 - Enhanced Call Forwarding Unconditional
 - Call Forwarding Busy/Don't Answer
 - Enhanced Call Forwarding Busy

If an application opens a stream with private data version 5 or later, and any monitor receives a Diverted event, then that Diverted Event is also sent to all other monitors associated with the call. A station device monitor, an ACD device monitor, or a call monitor can determine whether a call is leaving or staying at a previously alerted device (for example, when a call goes to a coverage point) by the absence or presence of the Diverted Event. Note that this change only affects how the Diverted event is reported; for private data version 5 there is no private data change for the Diverted Event itself.

 NOTE:

This behavior only applies to streams opened with private data version 5 or later. If an application opens a stream with private data version 4 or earlier, it will not be affected by this change.

Service Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>connection</code>	[mandatory] Specifies the connection that was alerting.
<code>divertingDevice</code>	[optional – partially supported] Specifies the device from which the call was diverted.
<code>newDestination</code>	[optional – partially supported] Specifies the device to which the call was diverted. The <code>deviceIDStatus</code> may be <code>ID_NOT_KNOWN</code> .
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> indicates that the local connection state is unknown.
<code>cause</code>	[optional – supported] Specifies the cause for this event. For Avaya Communication Manager Releases earlier than 6.3.6, the following cause is supported:

- `EC_REDIRECTED` – The call has been redirected.

Beginning with Avaya Communication Manager Release 6.3.6, the following causes are also supported:

- `EC_CALL_FORWARD_ALWAYS` (CS3/25) – The call has been redirected due to Call Forwarding.
- `EC_CALL_FORWARD_BUSY` (CS3/26) – The call has received Call Coverage treatment for one of the following reasons:
 - the monitored station is busy
 - there was not an available Call Appearance to receive the call
 - no endpoints are registered to the extension number of the monitored station.
- `EC_CALL_FORWARD_NO_ANSWER` (CS3/28) – The call has received Call Coverage treatment because the monitored station did not answer the call within the administered number of rings.
- `EC_CALL_FORWARD` (CS3/31)- The call has received Call Coverage treatment because Cover All Calls is active at the monitored station.

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).

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `DeviceID` that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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_NETWORK_SIGNAL` 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.

`callingDevice`

Specifies the calling device. The following rules apply:

- For internal calls – the originator’s extension.
- For incoming calls over PRI facilities – the “calling number” from the ISDN SETUP message is specified. If the “calling number” is not available, then the assigned trunk identifier is provided instead.
- For incoming calls over non-PRI facilities – the calling party number is generally not available. The assigned trunk identifier is provided instead.
- The trunk identifier is specified only when the calling party number is not available.

**NOTE:**

The trunk identifier is a dynamic device identifier and it cannot be used to access a trunk in Communication Manager.

- For calls originated at a bridged call appearance – the principal’s extension is specified.

Detailed Information:

In addition to the information below, see the [Event Report Detailed Information](#) on page 877.

- New Destination

**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, in part, on how an application monitors the devices involved in the call. Experimentation may be required before an application can use this information.
- This parameter provides expected destination of the call as known by the TSAPI Service. For example:

- If a call is redirected as a result of the Deflect Call service, the TSAPI Service reports the `calledDevice` parameter from the service request as the new destination.
- If a call is redirected as a result of the Pickup Call service, the TSAPI Service reports the `calledDevice` parameter from the service request as the new destination.

In other call scenarios, the Diverted event may indicate that the new destination is not known.

- ASAI Link Version 7 provides improved support for the `newDestination` parameter. However, even when using ASAI Link Version 7, some call scenarios remain where the Diverted event will report that the new destination is not known. For example, when a call is redirected to a coverage answer group, the `deviceIDStatus` of the `newDestination` parameter in the Diverted event will be `ID_NOT_KNOWN`.

Syntax

```

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

/* CSTADivertedEvent */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_DIVERTED */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefId;
            union
            {
                CSTADivertedEvent_t  diverted;
            } u;
        } cstaUnsolicited;
    } event;
} 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_t SubjectDeviceID_t;
typedef ExtendedDeviceID_t CalledDeviceID_t;

```

Private Data Syntax

If private data accompanies a `CSTADivertedEvent`, then the private data is 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 `CSTADivertedEvent` does not deliver private data to the application. If the `acsGetEventBlock()` or `acsGetEventPoll()` request returns a Private Data length of 0, then no private data is provided with this event.

Private Data Version 12 and Later Syntax

The CSTA Diverted Event includes a private data event, `ATTDivertedEvent` for private data version 12 and later. The `ATTDivertedEvent` provides the `deviceHistory` and `callingDevice` private data parameters.

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

/* ATTDivertedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_DIVERTED */
    union
    {
        ATTDivertedEvent_t    divertedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTDivertedEvent_t {
    DeviceHistory_t    deviceHistory;
    CallingDeviceID_t    callingDevice;
} ATTDivertedEvent_t;

typedef struct DeviceHistory_t {
    unsigned int        count;            /* at most 1 */
    DeviceHistoryEntry_t    *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t            olddeviceID;
    CSTAEventCause_t        cause;
    ConnectionID_t        oldconnectionID;
} DeviceHistoryEntry_t;
```

Private Data Version 7-11 Syntax

The CSTA Diverted Event includes a private data event, `ATTV11DivertedEvent` for private data versions 7-11. The `ATTV11DivertedEvent` provides the `deviceHistory` private data parameter.

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

/* ATTV11DivertedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV11_DIVERTED */
    union
    {
        ATTV11DivertedEvent_t    v11divertedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV11DivertedEvent_t {
    DeviceHistory_t    deviceHistory;
} ATTV11DivertedEvent_t;

typedef struct DeviceHistory_t {
    unsigned int                count;          /* at most 1 */
    DeviceHistoryEntry_t        *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t                olddeviceID;
    CSTAEventCause_t          cause;
    ConnectionID_t            oldconnectionID;
} DeviceHistoryEntry_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 Send All Calls feature for a specific device. When the Send All Calls 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 is available.

NOTE:

For Avaya Communication Manager, the Do Not Disturb event is only provided when the Send All Calls feature is activated or deactivated. This event is **not** provided when the Do Not Disturb feature is activated or deactivated.

Service Parameters

<code>acsHandle</code>	This is the handle for the ACS Stream.
<code>eventClass</code>	This is a tag with the value <code>CSTAUNSOLICITED</code> , which identifies this message as an CSTA unsolicited event.
<code>eventType</code>	This is a tag with the value <code>CSTA_DO_NOT_DISTURB</code> , which identifies this message as a <code>CSTADoNotDisturbEvent</code> .
<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>device</code>	[mandatory] Specifies the device for which the Send All Calls 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
<code>doNotDisturbOn</code>	[mandatory] Specifies whether the Send All Calls feature is on (1) or off (0).

Syntax

The following structure shows only the relevant portions of the unions for this message. See [ACS Data Types](#) on page 115 and [CSTA Event Data Types](#) on page 136 for a complete description of the event structure.

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

typedef struct
{
    ACSHandle_t          acsHandle;
    EventClass_t         eventClass;          /* CSTAUNSOLICITED */
    EventType_t         eventType;          /* CSTA_DO_NOT_DISTURB */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t    eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefID;
            union
            {
                CSTADoNotDisturbEvent_t    doNotDisturb,
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct CSTADoNotDisturbEvent_t {
    SubjectDeviceID_t    device;
    unsigned char        doNotDisturbOn;
} CSTADoNotDisturbEvent_t;
```

Endpoint Registered Event (Private Data Version 11 and later)

Summary

- Direction: Switch to Client
- Event: `CSTAPrivateStatusEvent`
- Private Data Event: `ATTEndpointRegisteredEvent`
- Service Parameters: `monitorCrossRefID`
- Private Parameters: `device`, `serviceState`, `endpointInfo`

Functional Description:

The Endpoint Registered event is sent when an H.323 or SIP endpoint registers to a monitored station extension.

This event is available beginning with Communication Manager Release 6.3.2 and AE Services Release 6.3.1.

This event is only available if the TSAPI Link is administered with ASAI Link Version 6 or later and if the application has negotiated private data version 11 or later.

Applications should use the `cstaGetAPICaps()` service and check the value of the `endpointRegisteredEvent` parameter in the private data accompanying the CSTA Get API Caps Confirmation event to determine whether this event will be provided.

Service Parameters:

`monitorCrossRefID` [mandatory] Contains the handle to the monitor request for which this event is reported.

Private Parameters:

`device` [mandatory] The extension number of the station.

`serviceState` [mandatory] The service state of the station extension.
The possible values are:

- `SS_IN_SERVICE` – The station is in service
- `SS_OUT_OF_SERVICE` – The station is out of service.
- `SS_UNKNOWN` – The station's service state is unknown.

`endpointInfo` [mandatory] A structure providing information about the H.323 or SIP endpoint that has registered to the station extension. This structure includes the following fields:

- `instanceID` – For H.323 endpoints registered through DMCC, the `instanceID` is 0-2. For H.323 endpoints not registered through DMCC and for SIP endpoints, the `instanceID` is always 0. To uniquely identify an endpoint,

applications must use both the `endpointAddress` and `instanceID` fields.

- `endpointAddress` – For H.323 endpoints, this is the IP address of the endpoint. For SIP endpoints, this is the endpoint's Universal Resource Identifier (URI).
- `switchEndIpAddress` – The switch-end IP address serving the endpoint.
- `macAddress` – The Media Access Control (MAC) address received from the endpoint when the endpoint registered.
- `productID` – For H.323 endpoints, this is an identifier submitted by the endpoint during registration. Its value is one of the product IDs administered on the Avaya Communication Manager system-parameters customer-options screen. For SIP endpoints, the `productID` is "SIP_Phone".
- `networkRegion` – The network region (1-250) administered for the endpoint on Avaya Communication Manager.
- `mediaMode` – The media mode in use by the endpoint. The possible values are:
 - `MM_CLIENT_SERVER` – The endpoint is registered in either client media mode or server media mode.
 - `MM_TELECOMMUTER` – The endpoint is registered in Telecommuter media mode.
 - `MM_NONE` – The endpoint is registered without media control. This media mode is sometimes referred to as "Shared Control" because it allows a DMCC application to share control of an extension with another endpoint registered to that extension.
 - `MM_OTHER` – The endpoint is registered with some other media mode not listed above.
- `dependencyMode` – The dependency mode in use by the endpoint. The possible values are:
 - `DM_MAIN` – The endpoint is registered with dependency mode Main. The endpoint can originate and receive calls. Only one endpoint can be registered to the extension with dependency mode Main. Typically, this is a physical set or an IP softphone.
 - `DM_DEPENDENT` – The endpoint is registered with dependency mode Dependent. An endpoint can only register with this dependency mode if another endpoint is already registered with dependency mode Main.
 - `DM_INDEPENDENT` – The endpoint is registered with dependency mode Independent. The endpoint can originate and receive calls even if another endpoint is not registered with dependency mode Main.

- `DM_OTHER` – The endpoint is registered with some other dependency mode not listed above.
- `unicodeScript` – For H.323 endpoints, this is a set of bit flags indicating which Unicode character sets are supported by the station. For SIP endpoints, this parameter is set to `US_NONE`.
- `stationType` – The station type administered for the extension.
- `signalingProtocol` – The signaling protocol for the endpoint. The possible values are:
 - `SP_H323` – The endpoint registered as an H.323 endpoint.
 - `SP_SIP` – The endpoint registered as a SIP endpoint.
 - `SP_NOT_SPECIFIED` – Avaya Communication Manager cannot provide the endpoint's signaling protocol.

Syntax

```

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

/* CSTAPrivateStatusEvent */

typedef struct {
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_PRIVATE_STATUS */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefId;
            union
            {
                CSTAPrivateEvent_t      privateStatus;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct CSTAPrivateEvent_t {
    Nulltype      null;
} CSTAPrivateEvent_t;

```

Private Data Version 11 and Later 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>

/* ATTEndpointRegisteredEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_ENDPOINT_REGISTERED */
    union
    {
        ATTEndpointRegisteredEvent_t    endpointRegisteredEvent;
    } u;
} ATTEvent_t;

typedef struct ATTEndpointRegisteredEvent_t {
    DeviceID_t                device;
    ATTSERVICESTATE_t         serviceState;
    ATTRegisteredEndpointInfo_t    endpointInfo;
} ATTEndpointRegisteredEvent_t;

typedef enum ATTSERVICESTATE_t {
    SS_UNKNOWN = -1,
    SS_OUT_OF_SERVICE = 0,
    SS_IN_SERVICE = 1
} ATTSERVICESTATE_t;

typedef struct ATTRegisteredEndpointInfo_t {
    ATTEndpointInstanceID_t    instanceID;
    ATTEndpointAddress_t        endpointAddress;
    ATTIPAddress_t              switchEndIpAddress;
    ATTMACAddress_t             macAddress;
    ATTPRODUCTID_t              productID;
    short                        networkRegion;
    ATTMediaMode_t              mediaMode;
    ATTDependencyMode_t         dependencyMode;
    ATTUnicodeScript_t          unicodeScript;
    ATTStationType_t            stationType;
    ATTSignalingProtocol_t      signalingProtocol;
} ATTRegisteredEndpointInfo_t;

typedef unsigned short    ATTEndpointInstanceID_t;

typedef char                ATTEndpointAddress_t[256];
```

Endpoint Registered Event (Private Data Version 11 and later)

```

typedef char                ATTIPAddress_t[46];

typedef char                ATTMACAddress_t[18];

typedef char                ATTProductID_t[16];

typedef enum ATTMediaMode_t {
    MM_OTHER = -1,
    MM_NONE = 0,
    MM_CLIENT_SERVER = 1,
    MM_TELECOMMUTER = 2
} ATTMediaMode_t;

typedef enum ATTDependencyMode_t {
    DM_OTHER = -1,
    DM_MAIN = 1,
    DM_DEPENDENT = 2,
    DM_INDEPENDENT = 3
} ATTDependencyMode_t;

typedef unsigned int        ATTUnicodeScript_t;

#define US_NONE              0x00000000
#define US_BASIC_LATIN      0x00000001
#define US_LATIN_1_SUPPLEMENT 0x00000002
#define US_LATIN_EXTENDED_A 0x00000004
#define US_LATIN_EXTENDED_B 0x00000008
#define US_GREEK_COPTIC     0x00000010
#define US_CYRILLIC          0x00000020
#define US_ARMENIAN          0x00000040
#define US_HEBREW            0x00000080
#define US_ARABIC            0x00000100
#define US_DEVANAGARI        0x00000200
#define US_BENGALI           0x00000400
#define US_GURMUKHI          0x00000800
#define US_GUJARATI          0x00001000
#define US_ORIYA             0x00002000
#define US_TAMIL             0x00004000
#define US_TELUGU            0x00008000
#define US_KANNADA           0x00010000

```

```

#define US_MALAYALAM
    0x00020000
#define US_SINHALA
    0x00040000
#define US_THAI
    0x00080000
#define US_LAO
    0x00100000
#define US_MYANMAR
    0x00200000
#define US_GEORGIAN
    0x00400000
#define US_TAGALOG
    0x00800000
#define US_KHMER
    0x01000000
#define US_HALF_WIDTH_AND_FULL_WIDTH_KANA      0x02000000
#define US_CJK_RADICALS_SUPPLEMENT_JAPAN      0x04000000
#define US_CJK_RADICALS_SUPPLEMENT_CHINA_S     0x08000000
#define US_CJK_RADICALS_SUPPLEMENT_CHINA_T     0x10000000
#define US_CJK_RADICALS_SUPPLEMENT_KOREAN      0x20000000
#define US_CJK_RADICALS_SUPPLEMENT_VIETNAM     0x40000000
#define US_HANGUL_JAMO
    0x80000000

typedef char                                ATTStationType_t[16];

typedef enum ATTSignalingProtocol_t {
    SP_NOT_PROVIDED = -1,
    SP_H323 = 1,
    SP_SIP = 2
} ATTSignalingProtocol_t;

```

Endpoint Unregistered Event (Private Data Version 11 and later)

- Direction: Switch to Client
- Event: `CSTAPrivateStatusEvent`
- Private Data Event: `ATTEndpointUnregisteredEvent`
- Service Parameters: `monitorCrossRefID`
- Private Parameters: `device`, `serviceState`, `instanceID`, `endpointAddress`, `switchEndIpAddress`, `dependencyMode`, `stationType`, `signalingProtocol`, `reason`, `cmReason`

Functional Description:

The Endpoint Unregistered event is sent when an H.323 or SIP endpoint unregisters from a monitored station extension.

This event is available beginning with Communication Manager Release 6.3.2 and AE Services Release 6.3.1.

This event is only available if the TSAPI Link is administered with ASAI Link Version 6 or later and if the application has negotiated private data version 11 or later.

Applications should use the `cstaGetAPICaps()` service and check the value of the `endpointUnregisteredEvent` parameter in the private data accompanying the CSTA Get API Caps Confirmation event to determine whether this event will be provided.

Service Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.

Private Parameters:

<code>device</code>	[mandatory] The extension number of the station.
<code>serviceState</code>	[mandatory] The service state of the station extension. The possible values are: • <code>SS_IN_SERVICE</code> – The station is in service • <code>SS_OUT_OF_SERVICE</code> – The station is out of service. • <code>SS_UNKNOWN</code> – The station's service state is unknown.

<code>instanceID</code>	[mandatory] For H.323 endpoints that had registered through DMCC, the <code>instanceID</code> is 0-2. For H.323 endpoints that had not registered through DMCC and for SIP endpoints, the <code>instanceID</code> is always 0. To uniquely identify an endpoint, applications must use both the <code>endpointAddress</code> and <code>instanceID</code> fields.

endpointAddress	[mandatory] For H.323 endpoints, this is the IP address of the endpoint. For SIP endpoints, this is the endpoint's Universal Resource Identifier (URI).
switchEndIpAddress	[mandatory] The switch-end IP address that had been serving the endpoint.
dependencyMode	[mandatory] The dependency mode with which the endpoint had been registered. The possible values are: • <code>DM_MAIN</code> – The endpoint was registered with dependency mode Main. An endpoint registered with dependency mode Main can originate and receive calls. Only one endpoint can be registered to the extension with dependency mode Main. Typically, this is a physical set or an IP softphone. • <code>DM_DEPENDENT</code> – The endpoint was registered with dependency mode Dependent. An endpoint can only register with this dependency mode if another endpoint is already registered with dependency mode Main. • <code>DM_INDEPENDENT</code> – The endpoint was registered with dependency mode Independent. An endpoint registered with dependency mode Independent can originate and receive calls even if another endpoint is not registered with dependency mode Main. • <code>DM_OTHER</code> – The endpoint was registered with some other dependency mode not listed above.
stationType	[mandatory] The station type administered for the extension.
signalingProtocol	[mandatory] The signaling protocol for the endpoint. The possible values are: • <code>SP_H323</code> – The endpoint was registered as an H.323 endpoint. • <code>SP_SIP</code> – The endpoint was registered as a SIP endpoint. • <code>SP_NOT_SPECIFIED</code> – Avaya Communication Manager cannot provide the signaling protocol used by the endpoint.
reason	[mandatory] The reason that the endpoint unregistered. Refer to Table 19 on page 746 for the possible reason values that may be reported in this parameter.
cmReason	[mandatory] The uninterpreted reason that the endpoint unregistered, as reported by Avaya Communication Manager.

Because future releases of Avaya Communication Manager may

include support for new reasons that an endpoint has unregistered, the uninterpreted value reported by Avaya Communication Manager is made directly available to applications. The value of the `reason` field for all such newly supported reasons will be `UR_OTHER`.

Refer to [Table 19](#) on page 746 for the known reason values that may be reported by Avaya Communication Manager in this parameter.

Table 19: Endpoint Unregistered Event Reasons

reason	cmReason	Description
UR_LOGOFF	0	The user logged off at the endpoint.
UR_NO_SIGNALING_- CONNECTION	2008	The signaling connection between CM and the endpoint has been lost.
UR_FORCE_LOGIN	2009	The endpoint was unregistered to allow a different endpoint to register at the same extension.
UR_SIGNALING_- CONNECTION_CLOSED	2010	The signaling connection for the endpoint has been closed.
UR_Q931_TIMEOUT	2011	CM did not receive a Q.931 SETUP message from the endpoint after it registered.
UR_TTL_EXPIRED	2012	The endpoint's time to live has expired without CM receiving a keep alive registration request.
UR_EXTENSION_REMOVED	2015	The extension where the endpoint was registered has been removed.
UR_SOFTPHONE	2016	The IP Softphone setting for the extension where the endpoint was registered has been changed from "y" to "n".
UR_NO_LAN_PORTS	2017	CM has run out of LAN port IDs.
UR_RO_TTI_MTC	2020	A background maintenance task has requested the removal of Remote Office LAN Terminal Translation Initialization (TTI) ports.
UR_SYSTEM_RESTART	2032	A system restart was initiated just after the endpoint registered.

Table 19: Endpoint Unregistered Event Reasons

UR_SAT_COMMAND	2033	A SAT command to unregister all H.323 endpoints has been invoked.
UR_CALL_EXPIRED	2077	The call preservation timer for the endpoint has expired.
UR_SHARED_CONTROL	1960	The endpoint is an IP soft phone in a shared control configuration with an IP telephone, and the telephone has unregistered.
UR_TSA	2080	The endpoint is an IP soft phone in a shared control configuration, and the user has entered telephone self administration mode.
UR_DCP_STN_STATE	2081	The endpoint is an IP soft phone in a shared control configuration with a DCP telephone, and the soft phone is no longer in a valid state.
UR_SHARED_CONTROL_MTC	2151	The endpoint is a shared control soft phone that is maintenance busy.
UR_MODULE_ID_MISMATCH	2175	Module ID mismatch.
UR_REGISTRATION_EXPIRED	2204	The endpoint's link loss delay timer has expired.
UR_INVALID_STATE	5037	The endpoint is in an invalid state.
UR_AUDIT_FAILURE	5061	A CM audit has detected a problem with the endpoint.
UR_REMOTE_MAX_MTC	2019	The endpoint is a Remote Max station that is undergoing maintenance.
UR_NO_H323_CONNECTION	2006	CM received a keep alive request from a dual connect endpoint whose H.323 extension is not registered.
UR_LOST_SIGNALING_CONNECTION	2007	The signaling connection has been lost for this endpoint.
UR_STATION_MTC	2018	The SAT commands "busyout station" and "release station" have been invoked for the station extension where the endpoint was registered.
UR_OTHER		A more specific reason cannot be provided.

Syntax

```

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

/* CSTAPrivateStatusEvent */

typedef struct {
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_PRIVATE_STATUS */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefId;
            union
            {
                CSTAPrivateEvent_t      privateStatus;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct CSTAPrivateEvent_t {
    Nulltype      null;
} CSTAPrivateEvent_t;

```

Private Data Version 11 and Later 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>

/* ATTEndpointRegisteredEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_ENDPOINT_UNREGISTERED */
    union
    {
        ATTEndpointUnregisteredEvent_t
        endpointUnregisteredEvent;
    } u;
} ATTEvent_t;

typedef struct ATTEndpointUnregisteredEvent_t {
    DeviceID_t                device;
    ATTSERVICESTATE_t         serviceState;
    ATTEndpointInstanceID_t   instanceID;
    ATTEndpointAddress_t      endpointAddress;
    ATTIPAddress_t            switchEndIpAddress;
    ATTDependencyMode_t       dependencyMode;
    ATTStationType_t          stationType;
    ATTSignalingProtocol_t     signalingProtocol;
    ATTUnregisterReason_t      reason;
    long                      cmReason;
} ATTEndpointUnregisteredEvent_t;

typedef enum ATTSERVICESTATE_t {
    SS_UNKNOWN = -1,
    SS_OUT_OF_SERVICE = 0,
    SS_IN_SERVICE = 1
} ATTSERVICESTATE_t;

typedef unsigned short  ATTEndpointInstanceID_t;

typedef char            ATTEndpointAddress_t[256];

typedef char            ATTIPAddress_t[46];

typedef enum ATTDependencyMode_t {
    DM_OTHER = -1,
```

```

    DM_MAIN = 1,
    DM_DEPENDENT = 2,
    DM_INDEPENDENT = 3
} ATTDependencyMode_t;

typedef char                               ATTStationType_t[16];

typedef enum ATTSignalingProtocol_t {
    SP_NOT_PROVIDED = -1,
    SP_H323 = 1,
    SP_SIP = 2
} ATTSignalingProtocol_t;

typedef enum ATTUnregisterReason_t {
    UR_OTHER = -1,
    UR_LOGOFF = 1,
    UR_NO_SIGNALING_CONNECTION = 2,
    UR_FORCE_LOGIN = 3,
    UR_SIGNALING_CONNECTION_CLOSED = 4,
    UR_Q931_TIMEOUT = 5,
    UR_TTL_EXPIRED = 6,
    UR_EXTENSION_REMOVED = 7,
    UR_SOFTPHONE = 8,
    UR_NO_LAN_PORTS = 9,
    UR_RO_TTI_MTC = 10,
    UR_SYSTEM_RESTART = 11,
    UR_SAT_COMMAND = 12,
    UR_CALL_EXPIRED = 13,
    UR_SHARED_CONTROL = 14,
    UR_TSA = 15,
    UR_DCP_STN_STATE = 16,
    UR_SHARED_CONTROL_MTC = 17,
    UR_MODULE_ID_MISMATCH = 18,
    UR_REGISTRATION_EXPIRED = 19,
    UR_INVALID_STATE = 20,
    UR_AUDIT_FAILURE = 21,
    UR_REMOTE_MAX_MTC = 22,
    UR_NO_H323_CONNECTION = 23,
    UR_LOST_SIGNALING_CONNECTION = 24,
    UR_STATION_MTC = 25
} ATTUnregisterReason_t;

```

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 is 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:

<code>connection</code>	[mandatory] Specifies the <code>callID</code> of the call for which this event is reported.
<code>digits</code>	[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.
<code>localConnectionInfo</code>	[optional] Specifies the local connection state as perceived by the monitored device on this call. A value of <code>CS_NONE</code> is always specified.
<code>cause</code>	[optional] Specifies the cause for this event.

Detailed Information:

See the [Event Report Detailed Information](#) on page 877.

Syntax

```

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

/* CSTAPrivateStatusEvent */

typedef struct {
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_PRIVATE_STATUS */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefId;
            union
            {
                CSTAPrivateEvent_t      privateStatus;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct CSTAPrivateEvent_t {
    Nulltype      null;
} CSTAPrivateEvent_t;

```

Private Data 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_t    eventType;          /* ATT_ENTERED_DIGITS */
    union
    {
        ATTEnteredDigitsEvent_t enteredDigitsEvent;
    } u;
} ATTEvent_t;

#define ATT_MAX_ENTERED_DIGITS 25

typedef struct ATTEnteredDigitsEvent_t {
    ConnectionID_t          connection;
    char
    digits[ATT_MAX_ENTERED_DIGITS];
    LocalConnectionState_t  localConnectionInfo;
    CSTAEventCause_t        cause;
} ATTEnteredDigitsEvent_t;
```

Established Event

Summary

- **Direction:** Switch to Client
- **Event:** CSTAEstablishedEvent
- **Private Data Event:** ATTEstablishedEvent (private data version 13), ATTV12EstablishedEvent (private data version 12), 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:** trunkGroup, trunkMember, split, lookaheadInfo, userEnteredCode, userInfo, reason, originalCallInfo, distributingDevice, distributingVDN, ucid, callOriginatorInfo, flexibleBilling, deviceHistory

Functional Description:

The Established Event Report indicates that the switch detects that a device has answered or connected to a call.



The Established Event Report is sent under the following circumstances:

- When a predictive 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 predictive call is placed to an off-PBX destination and an ISDN CONNECT message is received from an ISDN-PRI facility.
- When a predictive 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 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:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>establishedConnection</code>	[mandatory] Specifies the endpoint that joined the call.
<code>answeringDevice</code>	[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).
<code>callingDevice</code>	[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 – the "calling number" from the ISDN SETUP message is specified. If the "calling number" is not available, then the assigned trunk identifier is provided instead.

For incoming calls over PRI facilities – the "calling number" from the ISDN SETUP message is specified. If the "calling number" is not available, then the assigned trunk identifier is provided instead.

For incoming calls over non-PRI facilities – the calling party number is generally not available. The assigned trunk identifier is provided instead.

The trunk group number is specified only when the calling party number is not available.



NOTE:

The trunk identifier is a dynamic device identifier. It cannot be used to access a trunk in Communication Manager.

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 – 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 incoming 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 incoming calls over non-PRI facilities – the principal extension is specified. It may be a group extension for a TEG, hunt group, or 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.

<code>lastRedirectionDevice</code>	[optional – limited support] Specifies the previously redirection/alerted device in the case where the call was redirected/diverted to the <code>answeringDevice</code> .
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> indicates that the local connection state is unknown.
<code>cause</code>	[optional – supported] Specifies the cause for this event. The following causes are supported:

- `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 \(Private Data Version 5 and Later\)](#) in Chapter 6.
- `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 participant type `PT_SILENT`. For details, see [Single Step Conference Call Service \(Private Data Version 5 and Later\)](#) in Chapter 6.

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 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.

- `EC_PARK` – The call is connected due to picking up a parked call.
- `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. Beginning with Avaya Communication Manager Release 6.2, this includes the case where an EC500 user makes a call from his or her mobile phone and then bridges onto the call at their desk set.

This cause has higher precedence than `EC_NEW_CALL`.

- `EC_NEW_CALL` – The call has not yet been transferred.
- `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:

<code>trunkGroup</code>	[optional] Specifies the trunk group number from which the call originated. This parameter is supported by private data version 5 and later only.
<code>trunk</code>	[optional] Specifies the trunk group number from which the call originated. Trunk group number is provided only if the <code>callingDevice</code> is unavailable. This parameter is supported by private data versions 2, 3, and 4 only.
<code>trunkMember</code>	[optional – limited support] Specifies the trunk member number from which the call originated. This parameter is supported by private data version 5 and later.
<code>split</code>	[optional] Specifies the ACD split extension which delivered the call to an agent.
<code>distributingDevice</code>	[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. This parameter is supported by private data version 4 and later.
<code>distributingVDN</code>	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 a call is offered to the VDN. This event is conveyed to the applications as 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).
<code>lookaheadInfo</code>	[optional] Specifies the lookahead interflow information received from the established call. The 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 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 Communication Manager 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 Communication Manager 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 (private data versions 2-5) or 96 (private data versions 6 and later) bytes, with a call. This information may be a customer number, credit card number, alphanumeric digits, or a binary string.

**NOTE:**

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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` – indicates 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
- `AR_SERVICE_OBSERVER` (private data version 5 and later) – the Established Event Report resulted from a Service Observer (with either listen-only or listen-and-talk mode) joining the call.

`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`.

NOTE:

For the Established Event sent for 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 conference
 - `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` – the original `trunkMember` received by the `activeCall`. This parameter is supported by private data version 5 and later only.
- `lookaheadInfo` – the original `lookaheadInfo` received by the `activeCall`.
- `userEnteredCode` – the original `userEnteredCode` received by the `activeCall`.
- `userInfo` – the original `userInfo` received by the `activeCall`.

**NOTE:**

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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.

- `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 763.
- `olddeviceID`
- `cause`
- `oldconnectionID`

This parameter is supported by private data version 7 and later.

<code>ucid</code>	[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 <code>ucid</code> contains the <code>ATT_NULL_UCID</code> (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.
<code>callOriginatorInfo</code>	[optional] Specifies the <code>callOriginatorType</code> of the call originator such as coin call, 800-service call, or cellular call. See Table 18. Note: <code>callOriginatorType</code> values (II digit assignments) are provided by the network, not 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
<code>flexibleBilling</code>	[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 <code>TRUE</code>, 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.
<code>deviceHistory</code>	The <code>deviceHistory</code> parameter type specifies a list of <code>DeviceIDs</code> 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 <code>deviceHistory</code> 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).

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `DeviceID` that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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_NETWORK_SIGNAL` 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 below, see the [Event Report Detailed Information](#) on page 877.

- **Call Classification** – For `cstaMakePredictiveCall()` service requests, 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 41, which describes how to coordinate the passing of caller information across applications.
 - A cause of `EC_TRANSFER` indicates that an unsupervised transfer occurred before the call was established. An unsupervised 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:

During an unsupervised transfer, Communication Manager will not send a Transferred Event for the answering party, neither before nor 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 44. 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 Transferred 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_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_ESTABLISHED */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t monitorCrossRefId;
            union
            {
                CSTAEstablishedEvent_t  established;
            } u;
        } cstaUnsolicited;
    } event;
} 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 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.

Private Data Version 7 and Later Syntax

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
{
    ATTEventType_t    eventType;          /* ATT_ESTABLISHED */
    union
    {
        ATTEstablishedEvent_t    establishedEvent;
    } u;
} ATTEvent_t;

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;
    ATTChannelType_t        channelType;
} ATTEstablishedEvent_t;

typedef enum ATTChannelType_t {

    CH_OTHER = -1,
    CH_UNKNOWN = 0,
```

```

        CH_VOICE = 1,
        CH_VIDEO = 2
    } ATTChannelType_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 struct ATTUnicodeDeviceID_t {
    unsigned short                count;
    short                         value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE        25

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 provided */
    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;

#define ATT_MAX_USER_INFO      129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;          /* 0
indicates no UUI */
        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 */
} ATTUUIProtocolType_t;

typedef enum ATTReasonCode_t {
    AR_NONE = 0,                  /* no
reason code provided */
    AR_ANSWER_NORMAL = 1,         /* answer supervision
from the
    * network or internal answer */
    AR_ANSWER_TIMED = 2,          /* answer
assumed based on
    * internal timer */
    AR_ANSWER_VOICE_ENERGY = 3,   /* voice energy detection by
call
    * 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 */
} ATReasonCode_t

typedef struct ATOriginalCallInfo_t {
    ATReasonForCallInfo_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;
    unsigned char              flexibleBilling;
    DeviceHistory_t             deviceHistory;
} ATOriginalCallInfo_t;

typedef enum ATReasonForCallInfo_t {
    OR_NONE = 0,                /* indicates not present */
    OR_CONSULTATION = 1,
    OR_CONFERENCED = 2,
    OR_TRANSFERRED = 3,
    OR_NEW_CALL = 4
} ATReasonForCallInfo_t;

typedef ExtendedDeviceID_t      CallingDeviceID_t;
typedef ExtendedDeviceID_t      CalledDeviceID_t;

typedef char                    ATTUCID_t[64];

```

```

typedef struct ATTCallOriginatorInfo_t {
    unsigned char      hasInfo;          /* if FALSE, no
                                           *
call originator info */
    short              callOriginatorType;
} ATTCallOriginatorInfo_t;

typedef struct DeviceHistory_t {
    unsigned int        count;          /* at most 1 */
    DeviceHistoryEntry_t *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t          olddeviceID;
    CSTAEventCause_t    cause;
    ConnectionID_t      oldconnectionID;
} DeviceHistoryEntry_t;

```

Private Data Version 6 Syntax

For private data version 6, the maximum size of the data provided in the `userInfo` parameter is increased from 32 bytes to 96 bytes.

```
#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 ATTV6EstablishedEvent_t
{
    DeviceID_t                trunkGroup;
    DeviceID_t                trunkMember;
    DeviceID_t                split;
    ATTLookaheadInfo_t        lookaheadInfo;
    ATTUserEnteredCode_t      userEnteredCode;
    ATTUserToUserInfo_t       userInfo;
    ATTReasonCode_t           reason;
    ATTV6OriginalCallInfo_t    originalCallInfo;
    CalledDeviceID_t           distributingDevice;
    ATTUCID_t                 ucid;
    ATTCallOriginatorType_t    callOriginatorInfo;
    unsigned char              flexibleBilling;
} ATTV6EstablishedEvent_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 struct ATTUnicodeDeviceID_t {
    unsigned short    count;
    short            value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE    25

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 provided */
    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;

#define ATT_MAX_USER_INFO    129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t    type;
    struct
    {
        short                length;        /* 0
indicates no UUI */
        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 ATReasonCode_t {
    AR_NONE = 0, /* no
reason code provided */
    AR_ANSWER_NORMAL = 1, /* answer supervision
from the
    * network or internal answer */
    AR_ANSWER_TIMED = 2, /* answer
assumed based on
    * internal timer */
    AR_ANSWER_VOICE_ENERGY = 3, /* voice energy detection by
call
    * 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 */
} ATReasonCode_t

typedef struct ATTV6OriginalCallInfo_t {
    ATReasonForCallInfo_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;
    unsigned char            flexibleBilling;
} ATTV6OriginalCallInfo_t;

typedef enum ATReasonForCallInfo_t {

```

```

    OR_NONE = 0,                /* indicates not present */
    OR_CONSULTATION = 1,
    OR_CONFERENCED = 2,
    OR_TRANSFERRED = 3,
    OR_NEW_CALL = 4
} ATReasonForCallInfo_t;

typedef ExtendedDeviceID_t      CallingDeviceID_t;
typedef ExtendedDeviceID_t      CalledDeviceID_t;
typedef char                    ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char                hasInfo;          /* if FALSE, no
                                                    *
call originator info */
    short                       callOriginatorType;
} ATTCallOriginatorInfo_t;

```

Private Data Version 5 Syntax

Private data version 5 adds support for the `ucid`, `callOriginatorInfo`, `flexibleBilling`, and `uSourceVDN` parameters.

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

/* ATTV5EstablishedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV5_ESTABLISHED */
    union
    {
        ATTV5EstablishedEvent_t    v5establishedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV5EstablishedEvent_t
{
    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;
    unsigned char             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;

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 {
    unsigned short    count;
    short            value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE    25

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 provided */
    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 no UUI */
        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 ATTRReasonCode_t {

```



```

        AR_NONE = 0,                                /* no
reason code provided */
        AR_ANSWER_NORMAL = 1,                        /* answer supervision
from the
        * network or internal answer */
        AR_ANSWER_TIMED = 2,                        /* answer
assumed based on
        * internal timer */
        AR_ANSWER_VOICE_ENERGY = 3,                /* voice energy detection by
call
        * 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 */
} ATReasonCode_t

typedef struct ATTV5OriginalCallInfo_t {
    ATReasonForCallInfo_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;
    unsigned char            flexibleBilling;
} ATTV5OriginalCallInfo_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_t      CallingDeviceID_t;
typedef ExtendedDeviceID_t      CalledDeviceID_t;

typedef char                    ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char                hasInfo;                /* if FALSE, no
                                                         *
    call originator info */
    short                        callOriginatorType;
} ATTCallOriginatorInfo_t;

```

Private Data Version 4 Syntax

Private data version 4 adds support for the `distributingDevice` parameter.

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

/* ATTV4EstablishedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV4_ESTABLISHED */
    union
    {
        ATTV4EstablishedEvent_t    v4establishedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV4EstablishedEvent_t
{
    DeviceID_t                trunk;
    DeviceID_t                trunkMember;
    DeviceID_t                split;
    ATTV4LookaheadInfo_t      lookaheadInfo;
    ATTUserEnteredCode_t      userEnteredCode;
    ATTV5UserToUserInfo_t     userInfo;
    ATTReasonCode_t           reason;
    ATTV4OriginalCallInfo_t   originalCallInfo;
    CalledDeviceID_t          distributingDevice;
} ATTV4EstablishedEvent_t;

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;

#define ATT_MAX_USER_CODE      25

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 provided */
    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 no UUI */
        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 provided */
    AR_ANSWER_NORMAL = 1,                        /* answer supervision
from the
        * network or internal answer */
    AR_ANSWER_TIMED = 2,                        /* answer
assumed based on
        * internal timer */
    AR_ANSWER_VOICE_ENERGY = 3,                /* voice energy detection by
call
        * 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

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 ATReasonForCallInfo_t {
    OR_NONE = 0,                /* indicates not present */
    OR_CONSULTATION = 1,
    OR_CONFERENCED = 2,
    OR_TRANSFERRED = 3,
    OR_NEW_CALL = 4
} ATReasonForCallInfo_t;

typedef ExtendedDeviceID_t      CallingDeviceID_t;
typedef ExtendedDeviceID_t      CalledDeviceID_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_t    eventType;          /* ATTV3_ESTABLISHED */
    union
    {
        ATTV3EstablishedEvent_t    v3establishedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV3EstablishedEvent_t
{
    DeviceID_t                trunk;
    DeviceID_t                trunkMember;
    DeviceID_t                split;
    ATTV4LookaheadInfo_t      lookaheadInfo;
    ATTUserEnteredCode_t      userEnteredCode;
    ATTV5UserToUserInfo_t     userInfo;
    ATTReasonCode_t           reason;
    ATTV4OriginalCallInfo_t   originalCallInfo;
} ATTV3EstablishedEvent_t;

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;

#define ATT_MAX_USER_CODE      25

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 provided */
    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 no UUI */
        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 */
} ATTUUIProtocolType_t;

typedef enum ATTRReasonCode_t {

```



```

        AR_NONE = 0,                                /* no
reason code provided */
        AR_ANSWER_NORMAL = 1,                        /* answer supervision
from the
        * network or internal answer */
        AR_ANSWER_TIMED = 2,                          /* answer
assumed based on
        * internal timer */
        AR_ANSWER_VOICE_ENERGY = 3,                  /* voice energy detection by
call
        * 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 */
} ATReasonCode_t

typedef struct ATTV4OriginalCallInfo_t {
    ATReasonForCallInfo_t      reason;
    CallingDeviceID_t          callingDevice;
    CalledDeviceID_t           calledDevice;
    DeviceID_t                 trunk;
    DeviceID_t                 trunkMember;
    ATTV4LookaheadInfo_t       lookaheadInfo;
    ATTVUserEnteredCode_t      userEnteredCode;
    ATTV5UserToUserInfo_t      userInfo;
} ATTV4OriginalCallInfo_t;

typedef enum ATReasonForCallInfo_t {
    OR_NONE = 0,                                /* indicates not present */

```

```
    OR_CONSULTATION = 1,  
    OR_CONFERENCED = 2,  
    OR_TRANSFERRED = 3,  
    OR_NEW_CALL = 4  
} ATTRReasonForCallInfo_t;
```

```
typedef ExtendedDeviceID_t  
typedef ExtendedDeviceID_t
```

```
CallingDeviceID_t;  
CalledDeviceID_t;
```

Failed Event

Summary

- Direction: Switch to Client
- Event: CSTAFailedEvent
- Private Data Event: ATTFailedEvent (private data version 8 and later), 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.



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 is delivered to an on-PBX station without coverage where the Do Not Disturb feature is active.
- 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 at 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 an 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.
- When a Supervisor-Assist call is placed from an agent who is not logged in.

The Failed Event Report is **not** sent under the following circumstances:

- For a predictive call when any of the above conditions occur. In that case, a 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.

For ASAI Link Versions 1-6, the Failed Event is only sent to monitors for the calling device.

Beginning with ASAI Link Version 7, the Failed Event is also sent to monitors for the destination device.



NOTE:

According to the CSTA specification, a CSTA Failed event sent to a monitor for the destination device should be followed by a CSTA Connection Cleared event. However, the TSAPI Service does not send a CSTA Connection Cleared event for the failed connection to monitors for the destination device.

Service Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>failingConnection</code>	[mandatory – partially supported] Specifies the <code>callID</code> and <code>deviceID</code> of the failed connection. • Beginning with private data version 11, in most call scenarios the <code>deviceID</code> is the empty string (""). • For private data versions 2-10 (and when the application is not using private data), in most call scenarios the <code>deviceID</code> matches the <code>calledDevice</code>. This <code>deviceID</code> may be incorrect if, for example, the call was forwarded from the <code>calledDevice</code> to a different device where the call failed.

<code>failingDevice</code>	[mandatory – partially supported] Specifies the device that failed. The <code>deviceIdStatus</code> may be <code>ID_NOT_KNOWN</code>. - Beginning with private data version 11, in most call scenarios the <code>deviceId</code> is the empty string ("") and the <code>deviceIdStatus</code> is <code>ID_NOT_KNOWN</code>. - For private data versions 2-10 (and when the application is not using private data), in most call scenarios the <code>deviceId</code> matches the <code>calledDevice</code>. This <code>deviceId</code> may be incorrect if, for example, the call was forwarded from the <code>calledDevice</code> to a different device where the call failed.
<code>calledDevice</code>	[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 <code>NULL</code>), the <code>deviceIdStatus</code> is <code>ID_NOT_KNOWN</code>. - For outgoing calls over non-PRI facilities, then the <code>deviceIdStatus</code> is <code>ID_NOT_KNOWN</code>. - 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 <code>deviceIdStatus</code> is <code>ID_NOT_KNOWN</code>. - 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 <code>deviceIdStatus</code> is <code>ID_NOT_KNOWN</code>.
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for the <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> indicates that the local connection state is unknown.
<code>cause</code>	[optional – supported] Specifies the reason for this event. The following Event Causes are explicitly sent from the switch: <code>EC_BUSY</code> – User is busy or queue is full. <code>EC_CALL_NOT_ANSWERED</code> – User is not responding. <code>EC_TRUNKS_BUSY</code> – No trunks are available.

- `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` – The call failed for one of the following reasons:
 - The queue is full.
 - For a direct-agent call – the agent is not a member of the split or the agent is not logged in.
 - For a supervisor-assist call – the agent is not logged in.
- `EC_INCOMPATIBLE_BEARER_SERVICE` – The call failed because the selected facility for the call did not have the proper bearer capability. This could occur, for example, if a data or video call was attempted on a trunk facility that is reserved for voice traffic.

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).

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `DeviceID` that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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_NETWORK_SIGNAL` 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.

`callingDevice`

Specifies the calling device. The following rules apply:

- For internal calls – the originator's extension.
- For outgoing calls over PRI facilities – the "calling number" from the ISDN SETUP message is specified. If the "calling number" is not available, then the assigned trunk identifier is provided instead.
- For incoming calls over PRI facilities – the "calling number" from the ISDN SETUP message is specified. If the "calling number" is not available, then the assigned trunk identifier is provided instead.
- For incoming calls over non-PRI facilities – the calling party number is generally not available. The assigned trunk identifier is provided instead.
- The trunk identifier is specified only when the calling party number is not available.

**NOTE:**

The trunk identifier is a dynamic device identifier and it cannot be used to access a trunk in Communication Manager.

- For calls originated at a bridged call appearance – the principal's extension is specified.

Detailed Information:

See the [Event Report Detailed Information](#) on page 877.

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;
} 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 and Later Syntax

Private data version 8 adds support for the `callingDevice` parameter.

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

/* ATTFailedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_FAILED */
    union
    {
        ATTFailedEvent_t    failedEvent;
    } u;
} ATTEvent_t;

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

typedef struct DeviceHistory_t {
    unsigned int        count;            /* at most 1 */
    DeviceHistoryEntry_t    *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t            olddeviceID;
    CSTAEventCause_t        cause;
    ConnectionID_t        oldconnectionID;
} DeviceHistoryEntry_t;

typedef ExtendedDeviceID_t    CallingDeviceID_t;
```

Private Data Version 7 Syntax

The CSTA Failed Event includes a private data event, `ATTV7FailedEvent` for private data version 7. The `ATTV7FailedEvent` provides the `deviceHistory` private data parameter.

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

/* ATTV7FailedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV7_FAILED */
    union
    {
        ATTV7FailedEvent_t    v7failedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV7FailedEvent_t {
    DeviceHistory_t deviceHistory;
} ATTV7FailedEvent_t;

typedef struct DeviceHistory_t {
    unsigned int                count;          /* at most 1 */
    DeviceHistoryEntry_t        *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t                  olddeviceID;
    CSTAEventCause_t            cause;
    ConnectionID_t              oldconnectionID;
} DeviceHistoryEntry_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 is available.

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.

Service Parameters

<code>acsHandle</code>	This is the handle for the ACS Stream.
<code>eventClass</code>	This is a tag with the value <code>CSTAUNSOLICITED</code> , which identifies this message as an CSTA unsolicited event.
<code>eventType</code>	This is a tag with the value <code>CSTA_FORWARDING</code> which identifies this message as an <code>CSTAForwardingEvent</code> .
<code>monitorCrossRefID</code>	This parameter contains the handle to the CSTA association for which this event is associated. It provides a reference to a specific established association.
<code>device</code>	Specifies the device for which the Forwarding feature has been activated/deactivated.
<code>forwardingType</code>	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.
<code>forwardingOn</code>	Specifies whether the Forward feature is on (1) or off (0).
<code>forwardDN</code>	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 supports only the **Immediate** forwarding type.

**NOTE:**

AE Services always provides a null ("") value for this parameter.

Syntax

The following structure shows only the relevant portions of the unions for this message.

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

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_FORWARDING */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t      monitorCrossRefID;
            union
            {
                CSTAForwardingEvent_t    forwarding;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct CSTAForwardingEvent_t {
    SubjectDeviceID_t      device;
    ForwardingInfo_t        forwardingInformation;
} CSTAForwardingEvent_t;

typedef struct ForwardingInfo_t {
    ForwardingType_t        forwardingType;
    unsigned char           forwardingOn;
    DeviceID_t              forwardDN;          /* "" if not
present */
} ForwardingInfo_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;
```

Held Event

Summary

- Direction: Switch to Client
- Event: `CSTAHeldEvent`
- Private Data Event: `ATTHeldEvent` (private data version 9 and later)
- Service Parameters: `monitorCrossRefID`, `heldConnection`, `holdingDevice`, `localConnectionInfo`, `cause`
- Private Parameters: `consultMode`

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:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>heldConnection</code>	[mandatory] Specifies the endpoint where hold was activated.
<code>holdingDevice</code>	[mandatory] Specifies the station extension that placed the call on hold.
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> 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_NONE` – No cause provided.

Private Parameters:

consultMode

[optional – limited support] Indicates whether this event occurred as the result of a conference, transfer, or consultation operation. The following values are supported:

- `ATT_CM_NONE` – Indicates that the event did not occur as the result of a conference operation, a transfer operation, or Consultation Call service request.
- `ATT_CM_CONSULTATION` – Indicates that the event occurred as the result of a Consultation Call service request that is not part of a transfer or conference operation.
- `ATT_CM_TRANSFER` – Indicates that the event occurred as the result of a transfer operation.
- `ATT_CM_CONFERENCE` – Indicates that the event occurred as the result of a conference operation.
- `ATT_CM_NOT_PROVIDED` – Indicates that the TSAPI Service cannot determine why the event occurred.

This parameter is only supported by private data version 9 and later.

For private data version 9, the following limitations apply:

- The values `ATT_CM_NONE` and `ATT_CM_CONSULTATION` are not supported for this event.
- The values `ATT_CM_TRANSFER` and `ATT_CM_CONFERENCE` are only provided for manual transfer and conference operations performed at the telephone set, and only when:
 - the Communication Manager software release is 6.0.1 with service pack 1 or later
 - the TSAPI CTI link is administered with ASAI link version 5 or later.

Beginning with private data version 10, the `consultMode` is also provided when a call is placed on hold in response to a `cstaConsultationCall()` service request, ***but only on monitors for the holding device***:

- The value `ATT_CM_TRANSFER` is provided when a call is placed on hold using the `cstaConsultationCall()` service with consult option `CO_TRANSFER_ONLY`.
- The value `ATT_CM_CONFERENCE` is provided when a call is placed on hold using the `cstaConsultationCall()` service with consult option `CO_CONFERENCE_ONLY`.
- The value `ATT_CM_CONSULTATION` is provided when:
 - a call is placed on hold using the `cstaConsultationCall()` service with consult option `CO_CONSULT_ONLY` or `CO_UNRESTRICTED`
 - a call is placed on hold using the `cstaConsultationCall()` service with no consult options.

Detailed Information:

See the [Event Report Detailed Information](#) on page 877.

Syntax

```

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

/* CSTAHeldEvent */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;      /* CSTAUNSOLICITED */
    EventType_t      eventType;      /* CSTA_HELD */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t  monitorCrossRefId;
            union
            {
                CSTAHeldEvent_t      held;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct CSTAHeldEvent_t {
    ConnectionID_t      heldConnection;
    SubjectDeviceID_t   holdingDevice;
    LocalConnectionState_t  localConnectionInfo;
    CSTAEventCause_t    cause;
} CSTAHeldEvent_t;

```

Private Data Version 9 and Later Syntax

```

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

/* ATTHeldEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_HELD */
    union
    {
        ATTHeldEvent_t    heldEvent;
    } u;
} ATTEvent_t;

typedef struct ATTHeldEvent_t {
    ATTConsultMode_t    consultMode;
} ATTHeldEvent_t;

typedef enum ATTConsultMode_t {
    ATT_CM_NONE = 0,
    ATT_CM_CONSULTATION = 1,
    ATT_CM_TRANSFER = 2,
    ATT_CM_CONFERENCE = 3,
    ATT_CM_NOT_PROVIDED = 4
} ATTConsultMode_t;

```

Logged Off Event

Summary

- Direction: Switch to Client
- Event: `CSTALoggedOffEvent`
- Private Data Event: `ATTLoggedOffEvent`
- Service Parameters: `monitorCrossRefID`, `agentDevice`, `agentID`, `agentGroup`
- 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:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>agentDevice</code>	[mandatory] Indicates the extension of the agent that is logging off.
<code>agentID</code>	[optional – not supported] Indicates the agent identifier.
<code>agentGroup</code>	[optional – supported] Indicates the ACD Split that is being logged out of.

Private Parameters:

<code>reasonCode</code>	[optional] Specifies the reason that the agent changed to Auxiliary Work Mode (<code>WM_AUX_WORK</code>) or the logged-out (<code>AM_LOG_OUT</code>) state. Beginning with 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 only supported by private data version 7 and later. Private data versions 5 and 6 support 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 versions 4 and earlier do not support reason codes.

Detailed Information:

The CSTA Logged Off event is only provided when the Monitor Device service is used to monitor an ACD split. This event is not provided when the Monitor Device service is used to monitor a station extension.

See also the [Event Report Detailed Information](#) on page 877.

Syntax

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

/* CSTALoggedOffEvent */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_LOGGED_OFF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t      monitorCrossRefId;
            union
            {
                CSTALoggedOffEvent_t      loggedOff;
            } u;
        } cstaUnsolicited;
    } event;
} 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 Data 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>

/* ATTLLoggedOffEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_LOGGED_OFF */
    union
    {
        ATTLLoggedOffEvent_t    loggedOff;
    } u;
} ATTEvent_t;

typedef struct ATTLLoggedOffEvent_t {
    long        reasonCode;          /* 0-99 for private data version 7
                                     * and later; 0-
9 for private data
                                     * versions 5
and 6. */
} ATTLLoggedOffEvent_t;
```

Logged On Event

Summary

- Direction: Switch to Client
- Event: CSTALoggedOnEvent
- Private Data Event: ATTV14LoggedOnEvent, ATTLoggedOnEvent
- Service Parameters: monitorCrossRefID, agentDevice, agentID, agentGroup, password
- Private Parameters: workMode , skillLevel, percentAllocation

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:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>agentDevice</code>	[mandatory] Indicates the station extension of the agent that is logging on.
<code>agentID</code>	[optional – partially supported] Indicates the logical agent identifier. This is only provided in an Expert Agent Selection (EAS) environment. In a traditional ACD environment, this parameter is not supported.
<code>agentGroup</code>	[optional – supported] Indicates the ACD Split that is being logged into.
<code>password</code>	[optional – not supported] Indicates the agent password for logging in.

Private Parameters:

<code>workMode</code>	[optional – supported] Specifies the initial work mode for the Agent as Auxiliary-Work Mode (<code>WM_AUX_WORK</code>), After-Call-Work Mode (<code>WM_AFT_CALL</code>), Auto-In Mode (<code>WM_AUTO_IN</code>), or Manual-In-Work Mode (<code>WM_MANUAL_IN</code>). This parameter is supported by private data version 10 and later.

`skillLevel` [optional] `skillLevel` is a structure of the type `ATTSkillLevel_t`

Within `skillLevel`:

- `type` – indicates the skill level type. This field is set to `SKILL_LEVEL_TYPE_REGULAR` if the level provided is a normal skill level. This field is set to `SKILL_LEVEL_TYPE_RESERVED` if the level provided is a reserved skill level. If no skill level information is provided, this field is set to `SKILL_LEVEL_TYPE_NOT_PROVIDED`.
- `level` – indicates the skill level based on the type. If `type` is set to `SKILL_LEVEL_TYPE_NOT_PROVIDED`, this field is undefined and should not be used.

This parameter is only populated when the event is received on device monitor for skill.

`percentAllocation`

[optional] This parameter indicates the agent's percent allocation, if configured on CM, and can range from 1 to 100. If not configured on CM, or if not received from CM, it will default to 0.

This parameter is only populated when the event is received on device monitor for skill.

Detailed Information:

The CSTA Logged On event is only provided when the Monitor Device service is used to monitor an ACD split. This event is not provided when the Monitor Device service is used to monitor a station extension.

See also the [Event Report Detailed Information](#) on page 877.

Syntax

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

/* CSTALoggedOnEvent */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_LOGGED_ON */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
```

```

        {
            CSTAMonitorCrossRefID_t        monitorCrossRefId;
            union
            {
                CSTALoggedOnEvent_t        loggedOn;
            } u;
        } cstaUnsolicited;
    } event;
} 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 Data version 15 and later

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>

/* ATTLLoggedOnEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_LOGGED_ON */
    union
    {
        ATTLLoggedOnEvent_t    v15loggedOnEvent;
    } u;
} ATTEvent_t;

typedef struct ATTLLoggedOnEvent_t {
    ATTWorkMode_t        workMode;
    ATTSkillLevel_t      skillLevel;
    unsigned char        percentAllocation;
} ATTLLoggedOnEvent_t;

typedef enum ATTWorkMode_t {
    WM_NONE = -1,
    WM_AUX_WORK = 1,
    WM_AFTCAL_WK = 2,
    WM_AUTO_IN = 3,
    WM_MANUAL_IN = 4
} ATTWorkMode_t;

typedef struct ATTSkillLevel_t {
    ATTSkillLevelType_t    type;
    long                   level;
} ATTSkillLevel_t;

typedef enum ATTSkillLevelType_t {
    skLvlTypeNotProvided = 0,
    skLvlTypeRegular = 1,
    skLvlTypeReserved = 2
} ATTSkillLevelType_t;
```

Private Data Version 14

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>

/* ATTV14LoggedOnEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_LOGGED_ON */
    union
    {
        ATTV14LoggedOnEvent_t    loggedOnEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV14LoggedOnEvent_t {
    ATTWorkMode_t    workMode;
} ATTLoggedOnEvent_t;

typedef enum ATTWorkMode_t {
    WM_NONE = -1,
    WM_AUX_WORK = 1,
    WM_AFTCAL_WK = 2,
    WM_AUTO_IN = 3,
    WM_MANUAL_IN = 4
} ATTWorkMode_t;
```

Message Waiting Event

Summary

- **Direction:** Switch to Client
- **Event:** CSTAMessageWaitingEvent
- **Service Parameters:** monitorCrossRefID, deviceForMessage, invoking-Device, messageWaitingOn

Functional Description

This event report indicates a change in the status of the Message Waiting indicator for a specific device.

The Message Waiting event is available beginning with AE Services Release 6.3.3 and Communication Manager Release 6.3.6. 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 is available.

Service Parameters

<code>acsHandle</code>	This is the handle for the ACS Stream.
<code>eventClass</code>	This is a tag with the value <code>CSTAUNSOLICITED</code> , which identifies this message as an CSTA unsolicited event.
<code>eventType</code>	This is a tag with the value <code>CSTA_MESSAGE_WAITING</code> , which identifies this message as a <code>CSTAMessageWaitingEvent</code> .
<code>monitorCrossRefID</code>	[mandatory] This parameter contains the handle to the monitor request for which this event is reported.
<code>deviceForMessage</code>	[mandatory] This parameter specifies the device for which the Message Waiting indicator has been activated or deactivated.
<code>invokingDevice</code>	[mandatory – not supported] This parameter specifies the device which activated or deactivated the Message Waiting feature. The <code>deviceID</code> is set to the empty string (“”) and the <code>deviceIDStatus</code> is reported as <code>ID_NOT_KNOWN</code> .
<code>messageWaitingOn</code>	[mandatory] This parameter specifies whether the Message Waiting indicator is on (1) or off (0).

Detailed Information:

- Application Type – Multiple applications may turn on a station's Message Waiting indicator. These applications include: Property Management, Message Center, Voice Messaging, Leave Word Calling (LWC), and CTI.

The CSTA Message Waiting event is only provided when the status of the Message Waiting indicator changes from on to off or from off to on. For example, if the Message Waiting indicator has already been turned on for one application, the CSTA Message Waiting event is not provided if the Message Waiting indicator is then turned on for a second application.

An application may use the `cstaQueryMsgWaitingInd()` service to determine for which application types the Message Waiting indicator is on.

Syntax

The following structure shows only the relevant portions of the unions for this message. See [ACS Data Types](#) on page 115 and [CSTA Event Data Types](#) on page 136 for a complete description of the event structure.

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

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_MESSAGE_WAITING */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t      monitorCrossRefID;
            union
            {
                CSTAMessageWaitingEvent_t messageWaiting,
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

typedef struct CSTAMessageWaitingEvent_t {
    CalledDeviceID_t      deviceForMessage;
    SubjectDeviceID_t     invokingDevice;
    unsigned char         messageWaitingOn;
} CSTAMessageWaitingEvent_t;
```

Network Reached Event

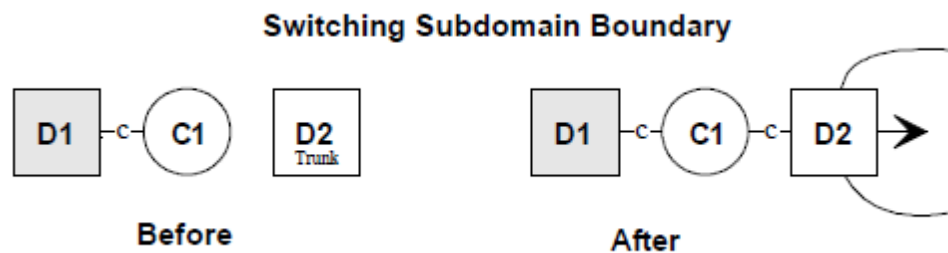
Summary

- **Direction:** Switch to Client
- **Event:** CSTANetworkReachedEvent
- **Private Data Event:** ATTNetworkReachedEvent (private data version 7 and later), ATTV6NetworkReachedEvent (private data versions 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 predictive call.

Service Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>connection</code>	[mandatory] Specifies the endpoint for the outbound connection to another network.
<code>trunkUsed</code>	[mandatory – not supported] Specifies the trunk that was used to establish the connection. This information is provided in the private data.
<code>calledDevice</code>	[mandatory – partially supported] Specifies the destination device of the call. The <code>deviceIDStatus</code> may be <code>ID_NOT_KNOWN</code> .
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for <code>cstaMonitorDevice</code> requests only. A value of <code>CS_NONE</code> indicates that the local connection state is unknown.
<code>cause</code>	[optional – supported] Specifies the cause for this event. The following causes are supported: • <code>EC_REDIRECTED</code> – The call has been redirected. • <code>EC_SINGLE_STEP_TRANSFER</code> (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_NONE` – not provided
- `PL_USER` – user
- `PL_PUB_LOCAL` – public network serving local user
- `PL_PUB_REMOTE` – public network serving remote user
- `PL_PRIV_REMOTE` – private network serving remote user

`progressDescription`

[mandatory] Specifies the progress description in a Progress Indicator Information Element from the PRI network. The following description indicators are supported:

- `PD_NONE` – not provided
- `PD_CALL_OFF_ISDN` – the call is not end-to-end ISDN, call progress in-band
- `PD_DEST_NOT_ISDN` – the destination address is non-ISDN
- `PD_ORIG_NOT_ISDN` – the origination address is no-ISDN
- `PD_CALL_ON_ISDN` – the call has returned to ISDN
- `PD_INBAND` – in-band information is now available

`trunkGroup`

[optional – limited support] 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 support] 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).

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `deviceID` that had previously been associated with the call and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the `cause` and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the `cause` provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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_NETWORK_SIGNAL` 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:

See the [Event Report Detailed Information](#) on page 877.

Syntax

```

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

/* CSTANetworkReachedEvent */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_NETWORK_REACHED */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t      monitorCrossRefId;
            union
            {
                CSTANetworkReachedEvent_t  networkReached;
            } u;
        } cstaUnsolicited;
    } event;
} 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 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.

Private Data Version 7 and Later Syntax

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

/* ATTNetworkReachedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_NETWORK_REACHED */
    union
    {
        ATTNetworkReachedEvent_t    networkReachedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTNetworkReachedEvent_t {
    ATTPProgressLocation_t    progressLocation;
    ATTPProgressDescription_t    progressDescription;
    DeviceID_t                trunkGroup;
    DeviceID_t                trunkMember;
    DeviceHistory_t            deviceHistory;
} ATTNetworkReachedEvent_t;

typedef enum ATTPProgressLocation_t {
    PL_NONE = -1,                /* not provided */
    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 */
} ATTPProgressLocation_t;

typedef enum ATTPProgressDescription_t {
    PD_NONE = -1,                /* not provided */
    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 */
} ATTPProgressDescription_t;
```

Private Data Version 5 and 6 Syntax

```

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

/* ATTV6NetworkReachedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV6_NETWORK_REACHED */
    union
    {
        ATTV6NetworkReachedEvent_t    v6networkReachedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV6NetworkReachedEvent_t
{
    ATTProgressLocation_t    progressLocation;
    ATTProgressDescription_t    progressDescription;
    DeviceID_t                trunkGroup;
    DeviceID_t                trunkMember;
} ATTV6NetworkReachedEvent_t;

typedef enum ATTProgressLocation_t {
    PL_NONE = -1,                /* not provided */
    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;

typedef enum ATTProgressDescription_t {
    PD_NONE = -1,                /* not provided */
    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_t    eventType;          /* ATTV4_NETWORK_REACHED */
    union
    {
        ATTV4NetworkReachedEvent_t    v4networkReachedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV4NetworkReachedEvent_t {
    ATTProgressLocation_t    progressLocation;
    ATTProgressDescription_t    progressDescription;
} ATTV4NetworkReachedEvent_t;

typedef enum ATTProgressLocation_t {
    PL_NONE = -1,                /* not provided */
    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;

typedef enum ATTProgressDescription_t {
    PD_NONE = -1,                /* not provided */
    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: ATTOrganizedEvent (private data version 15 and later), ATT14OrganizedEvent (private data version 9-14), ATTV8OrganizedEvent (private data versions 6-8), ATTV5OrganizedEvent (private data versions 2-5)
- Service Parameters: monitorCrossRefID, originatedConnection, callingDevice, calledDevice, localConnectionInfo, cause
- Private Parameters: logicalAgent, userInfo, consultMode, channelType

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.



This event is generated under the following circumstances:

- 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:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>originatedConnection</code>	[mandatory] Specifies the connection for which the call has been originated.
<code>callingDevice</code>	[mandatory] Specifies the device from which the call has been originated.
<code>calledDevice</code>	[mandatory] Specifies the number that the user dialed or the destination requested by a <code>cstaMakeCall</code> . 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).
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This information is provided for <code>cstaMonitorDevice</code> requests only. A value of <code>CS_NONE</code> indicates that the local connection state is unknown.
<code>cause</code>	[optional – supported] Specifies the cause for this event. The following causes are supported: • <code>EC_KEY_CONFERENCE</code> – Indicates that the event report occurred at a bridged device. This cause has higher precedence than the following cause. • <code>EC_NEW_CALL</code> – The event report is for a new call.

Private Parameters:

<code>logicalAgent</code>	[optional] Specifies the logical agent extension of the agent that is logged into the station making the call for a <code>cstaMakeCall()</code> request.
<code>userInfo</code>	[optional – not supported] This parameter allows the application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) bytes, with a call. This information may be a customer number, credit card number, alphanumeric digits, or a binary string. The <code>userInfo</code> parameter is defined for this event, but it is not supported by Communication Manager. Thus, the <code>userInfo</code> parameter will not be provided for this event.
<code>consultMode</code>	[optional – limited support] Indicates whether this event occurred as the result of a conference, transfer, or consultation operation. The following values are supported: • <code>ATT_CM_NONE</code> – Indicates that the event did not occur as the result of a conference operation, a transfer operation, or Consultation Call service request. • <code>ATT_CM_CONSULTATION</code> – Indicates that the event occurred as the result of a Consultation Call service request that is not part of a transfer or conference operation. • <code>ATT_CM_TRANSFER</code> – Indicates that the event occurred as the result of a transfer operation. • <code>ATT_CM_CONFERANCE</code> – Indicates that the event occurred as the result of a conference operation. • <code>ATT_CM_NOT_PROVIDED</code> – Indicates that the TSAPI Service cannot determine why the event occurred.

This parameter is only supported by private data version 9 and later.

For private data version 9, the following limitations apply:

- The value `ATT_CM_NONE` is not supported for this event.
- The value `ATT_CM_CONSULTATION` is provided when a call is originated using the `cstaConsultationCall()` service.
- The values `ATT_CM_TRANSFER` and `ATT_CM_CONFERERENCE` are only provided for manual transfer and conference operations performed at the telephone set, and only when:
 - the Communication Manager software release is 6.0.1 with service pack 1 or later
 - the TSAPI CTI link is administered with ASAI link version 5 or later.

Beginning with private data version 10:

- The value `ATT_CM_TRANSFER` is also provided when a call is originated using the `cstaConsultationCall()` service with consult option `CO_TRANSFER_ONLY`.
- The value `ATT_CM_CONFERERENCE` is also provided when a call is originated using the `cstaConsultationCall()` service with consult option `CO_CONFERERENCE_ONLY`.
- The value `ATT_CM_CONSULTATION` is provided when:
 - a call is originated using the `cstaConsultationCall()` service with consult option `CO_CONSULT_ONLY` or `CO_UNRESTRICTED`
 - a call is originated using the `cstaConsultationCall()` service with no consult options.

`channelType`

[optional] Specifies type of call. Means if call is voice than its value is `CH_VOICE`, if call is type of video its value is `CH_VIDEO`.

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 request.

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_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t      monitorCrossRefId;
            union
            {
                CSTAOriginatedEvent_t    originated;
            } u;
        } cstaUnsolicited;
    } event;
} 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 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.

Private Data Version 15 and Later Syntax

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

/* ATTOrganizedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_ORIGINAL */
    union
    {
        ATTOrganizedEvent_t    originatedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTOrganizedEvent_t {
    DeviceID_t          logicalAgent,
    ATTUserToUserInfo_t    userInfo,
    ATTConsultMode_t      consultMode,
    ATTChannelType_t      channelType
} ATTOrganizedEvent_t;

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t    type;
    struct
    {
        short                                length;          /* 0
indicates no UUI */
        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 ATTConsultMode_t {
    ATT_CM_NONE = 0,
    ATT_CM_CONSULTATION = 1,
```

```

        ATT_CM_TRANSFER = 2,
        ATT_CM_CONFERENCE = 3,
        ATT_CM_NOT_PROVIDED = 4
    } ATTConsultMode_t;

typedef enum ATTChannelType_t {
    CH_OTHER = -1,
    CH_UNKNOWN = 0,
    CH_VOICE = 1,
    CH_VIDEO = 2
} ATTChannelType_t;

```

Private Data Version 9-14 Syntax

```

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

/* ATT14OriginatedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_ORIGINATED */
    union
    {
        ATTOrganizedEvent_t    originatedEvent;
    } u;
} ATTEvent_t;

typedef struct ATT14OriginatedEvent_t {
    DeviceID_t          logicalAgent;
    ATTUserToUserInfo_t    userInfo;
    ATTConsultMode_t      consultMode;
} ATTOrganizedEvent_t;

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t    type;
    struct
    {
        short                                length;          /* 0
indicates no UUI */
        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 ATTConsultMode_t {
    ATT_CM_NONE = 0,
    ATT_CM_CONSULTATION = 1,
    ATT_CM_TRANSFER = 2,
    ATT_CM_CONFERENCE = 3,
    ATT_CM_NOT_PROVIDED = 4
} ATTConsultMode_t;
```

Private Data Version 6-8 Syntax

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

/* ATTV8OriginatedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV8_ORIGINATED */
    union
    {
        ATTV8OriginatedEvent_t  v8originatedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV8OriginatedEvent_t {
    DeviceID_t                logicalAgent;
    ATTUserToUserInfo_t       userInfo;
} ATTV8OriginatedEvent_t;

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;      /* 0
indicates no UUI */
        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>

/* ATTV5OriginatedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV5_ORIGINATED */
    union
    {
        ATTV5OriginatedEvent_t  v5originatedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV5OriginatedEvent_t {
    DeviceID_t                logicalAgent;
    ATTV5UserToUserInfo_t    userInfo;
} ATTV5OriginatedEvent_t;

typedef struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t    type;
    struct
    {
        short                length;      /* 0
indicates no UUI */
        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 and later)
- 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.



The Queued Event report is generated under the following circumstances:

- 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:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>queuedConnection</code>	[mandatory] Specifies the connection that queued.
<code>queue</code>	[mandatory] Specifies the queuing device to which the call has queued. This is the extension of the ACD split to which the call queued.
<code>callingDevice</code>	[mandatory – partially supported] Specifies the calling device. The <code>deviceIDStatus</code> may be <code>ID_NOT_KNOWN</code> .
<code>calledDevice</code>	[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 <code>deviceIDStatus</code> is <code>ID_NOT_KNOWN</code>. 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, the dialed number is specified.
<code>lastRedirectionDevice</code>	[optional – limited support] Specifies the previous redirection/alerted device in case where the call was redirected/diverted to the queue device.
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> indicates that the local connection state is unknown.
<code>numberQueued</code>	[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 causes are supported:

- `EC_REDIRECTED` – The call has been redirected.
- `EC_TRANSFER` (private data versions 2-7) – The call was queued as the result of a Single Step Transfer Call operation.
- `EC_SINGLE_STEP_TRANSFER` (private data version 8 or later) – The call was queued as the result of a Single Step Transfer Call operation.
- `EC_NEW_CALL` – The call entered the queue neither by being redirected/diverted to the queue, nor 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).

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `DeviceID` that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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_NETWORK_SIGNAL` 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](#) on page 877.

- 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 Avaya Communication Manager:

- For any vector controlled ACD or Skill (in either an Expert Agent Selection [EAS] or non-EAS environment) use `cstaMonitorCallsViaDevice()` to monitor the VDN that queues calls to the ACD or Skill.
- For a non-vector controlled ACD (in a non-EAS environment) use `cstaMonitorCallsViaDevice()` to monitor the ACD split.

Syntax

```

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

/* CSTAQueuedEvent */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;      /* CSTAUNSOLICITED */
    EventType_t      eventType;      /* CSTA_QUEUED */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t  monitorCrossRefId;
            union
            {
                CSTAQueuedEvent_t     queued;
            } u;
        } cstaUnsolicited;
    } event;
} 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 Later Syntax

The CSTA Queued Event includes a private data event, `ATTQueuedEvent` for private data version 7. The `ATTQueuedEvent` provides the `deviceHistory` private data parameter.

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

/* ATTQueuedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_QUEUED */
    union
    {
        ATTQueuedEvent_t    queuedEvent;
    } u;
} ATTEvent_t;

typedef struct ATTQueuedEvent_t {
    DeviceHistory_t    deviceHistory;
} ATTQueuedEvent_t;

typedef struct DeviceHistory_t {
    unsigned int        count;            /* at most 1 */
    DeviceHistoryEntry_t    *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t            olddeviceID;
    CSTAEventCause_t      cause;
    ConnectionID_t        oldconnectionID;
} DeviceHistoryEntry_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.



It is generated when an on-PBX station connects to a call that has been previously placed on hold. Retrieving 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:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>retrievedConnection</code>	[mandatory] Specifies the connection for which the call has been taken off the held state.
<code>retrievingDevice</code>	[mandatory] Specifies the device that connected the call from the held state. This is the extension that has been connected to the call.
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> 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_NONE` – Indicates that the event report did not occur at a bridged device.

Detailed Information:

See the [Event Report Detailed Information](#) on page 877.

Syntax

```

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

/* CSTARetrievedEvent */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_RETRIEVED */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t      monitorCrossRefId;
            union
            {
                CSTARetrievedEvent_t      retrieved;
            } u;
        } cstaUnsolicited;
    } event;
} 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` (private data version 9 and later), `ATTV8ServiceInitiatedEvent` (private data versions 5-8)
- Service Parameters: `monitorCrossRefID`, `initiatedConnection`, `localConnectionInfo`, `cause`
- Private Parameters: `ucid`

Functional Description:

The Service Initiated Event Report indicates that telecommunication service is initiated.



This event is generated under the following circumstances:

- 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 abbreviated dialing, etc.) are invoked.

Service Parameters:

<code>monitorCrossRefID</code>	[mandatory] Contains the handle to the monitor request for which this event is reported.
<code>initiatedConnection</code>	[mandatory] Specifies the connection for which the service (dial tone) has been initiated.
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for the <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> indicates that the local connection state is unknown.
<code>cause</code>	[optional – supported] Specifies the cause for this event. The following cause values are supported:

- `EC_KEY_CONFERENCE` – Indicates that the event report occurred at a bridged device.
- `EC_NONE` – Indicates that the event report did not occur 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.

`consultMode`

[optional – limited support] Indicates whether this event occurred as the result of a conference, transfer, or consultation operation. The following values are supported:

- `ATT_CM_NONE` – Indicates that the event did not occur as the result of a conference operation, a transfer operation, or Consultation Call service request.
- `ATT_CM_CONSULTATION` – Indicates that the event occurred as the result of a Consultation Call service request that is not part of a transfer or conference operation.
- `ATT_CM_TRANSFER` – Indicates that the event occurred as the result of a transfer operation.
- `ATT_CM_CONFERENCE` – Indicates that the event occurred as the result of a conference operation.
- `ATT_CM_NOT_PROVIDED` – Indicates that the TSAPI Service cannot determine why the event occurred.

This parameter is only supported by private data version 9 and later.

For private data version 9, the following limitations apply:

- The values `ATT_CM_NONE` and `ATT_CM_CONSULTATION` are not supported for this event.
- The values `ATT_CM_TRANSFER` and `ATT_CM_CONFERENCE` are only provided for manual transfer and conference operations performed at the telephone set, and only when:
 - the Communication Manager software release is 6.0.1 with service pack 1 or later
 - the TSAPI CTI link is administered with ASAI link version 5 or later.

Beginning with private data version 10, the `consultMode` is also provided when a call is initiated in response to a `cstaConsultationCall()` service request:

- The value `ATT_CM_TRANSFER` is provided when a call is initiated using the `cstaConsultationCall()` service with consult option `CO_TRANSFER_ONLY`.
- The value `ATT_CM_CONFERENCE` is provided when a call is initiated using the `cstaConsultationCall()` service with consult option `CO_CONFERENCE_ONLY`.
- The value `ATT_CM_CONSULTATION` is provided when:
 - a call is initiated using the `cstaConsultationCall()` service with consult option `CO_CONSULT_ONLY` or `CO_UNRESTRICTED`
 - a call is initiated using the `cstaConsultationCall()` service with no consult options.

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_t     eventClass;          /* CSTAUNSOLICITED */
    EventType_t      eventType;          /* CSTA_SERVICE_INITIATED */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            CSTAMonitorCrossRefID_t      monitorCrossRefId;
            union
            {
                CSTAServiceInitiatedEvent_t serviceInitiated;
            } u;
        } cstaUnsolicited;
    } event;
} CSTAEvent_t;

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

```

Private Data 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.

Private Data Version 9 and Later Syntax

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

/* ATTSERVICEInitiatedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;           /* ATT_SERVICE_INITIATED */
    union
    {
        ATTSERVICEInitiatedEvent_t    serviceInitiated;
    } u;
} ATTEvent_t;

typedef struct ATTSERVICEInitiatedEvent_t {
    ATTUCID_t            ucid;
    ATTConsultMode_t     consultMode;
} ATTSERVICEInitiatedEvent_t;

typedef char ATTUCID_t[64];

typedef enum ATTConsultMode_t {
    ATT_CM_NONE = 0,
    ATT_CM_CONSULTATION = 1,
    ATT_CM_TRANSFER = 2,
    ATT_CM_CONFERENCE = 3,
    ATT_CM_NOT_PROVIDED = 4
} ATTConsultMode_t;
```

Private Data Version 5-8 Syntax

```

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

/* ATTV8ServiceInitiatedEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV8_SERVICE_INITIATED
*/
    union
    {
        ATTV8ServiceInitiatedEvent_t  v8serviceInitiated;
    } u;
} ATTEvent_t;

typedef struct ATTV8ServiceInitiatedEvent_t {
    ATTUCID_t          ucid;
} ATTV8ServiceInitiatedEvent_t;

typedef char ATTUCID_t[64];

```

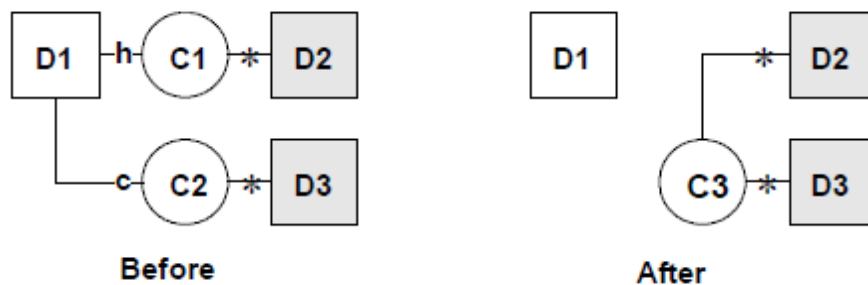
Transferred Event

Summary

- **Direction:** Switch to Client
- **Event:** `CSTATransferredEvent`
- **Private Data Event:** `ATTTransferredEvent` (private data version 7 and later), `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 under the following circumstances:

- When an on-PBX station completes a transfer by pressing the "transfer" button on the voice terminal.
- When an on-PBX analog telephone 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.

<code>primaryOldCall</code>	[mandatory] Specifies the <code>callID</code> of the call that was transferred. This is usually the held call from before the transfer. This call ended as a result of the transfer.
<code>secondaryOldCall</code>	[mandatory] Specifies the <code>callID</code> of the call that was transferred. This is usually the active call from before the transfer. This call is retained by the switch after the transfer.
<code>transferringDevice</code>	[mandatory] Specifies the device that is controlling the transfer. This is the device that did the transfer.
<code>transferredDevice</code>	[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 <code>deviceIDStatus</code> is <code>ID_NOT_KNOWN</code>. There are call scenarios in which the transfer operation joins multiple parties to a call. In such situations, the <code>transferredDevice</code> will be the extension for the last party to join the call.
<code>transferredConnections</code>	[optional – supported] Specifies a count of the number of devices and a list of <code>connectionIDs</code> and <code>deviceIDs</code> 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 <code>deviceID</code> 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.
<code>localConnectionInfo</code>	[optional – supported] Specifies the local connection state as perceived by the monitored device on this call. This is provided for the <code>cstaMonitorDevice()</code> requests only. A value of <code>CS_NONE</code> indicates that the local connection state is unknown.
<code>cause</code>	[optional – supported] Specifies the cause for this event. The following causes are supported: • <code>EC_TRANSFER</code> – A call transfer has occurred. • <code>EC_PARK</code> – 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.

In a `cstaTransferCall()` scenario, 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` – 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.
- For private data versions 2-5, the maximum length of `userInfo` is 32 bytes. Beginning with private data version 6, the maximum length of `userInfo` is increased to 96 bytes.
- An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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 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 858.
 - `olddeviceID`
 - `cause`
 - `oldconnectionID`

This parameter is supported by private data version 7 and later.

`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.

<code>distributingVDN</code>	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 a call is offered to the VDN. This event is conveyed to the applications as 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).
<code>ucid</code>	[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 <code>ucid</code> contains the <code>ATT_NULL_UCID</code> (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.
<code>trunkList</code>	[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: • <code>count</code> – The count of the connected parties on the call. • <code>trunks</code> – An array of 5 trunk group and trunk member IDs, one for each connected party. The following options are supported: – <code>connection</code> – The connection ID of one of the parties on the call. – <code>trunkGroup</code> – The trunk group of the party referenced by connection. – <code>trunkMember</code> – The trunk member of the party referenced by connection.
<code>deviceHistory</code>	The <code>deviceHistory</code> parameter type specifies a list of <code>DeviceIDs</code> 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 <code>deviceHistory</code> 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).

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `deviceID` that had previously been associated with the call and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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_NETWORK_SIGNAL` 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 the 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 that ended as the result of the `cstaTransferCall()` is usually the `heldCall`, sometimes it can be the `activeCall`.
- If the Transferred Event was the 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 that ended as the result of the manual transfer operation) is usually the `heldCall`, sometimes it can be the active call.

In addition, see the Established Event [Detailed Information](#) section for Unsupervised Transfer and Consultation Transfer definitions; [Transferring or conferencing a call together with screen pop information](#) on page 41 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;
} 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;

typedef ExtendedDeviceID_t    SubjectDeviceID_t;

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

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

```

Private Data 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.

Private Data Version 7 and Later Syntax

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 ATTEvent_t
{
    ATTEventType_t    eventType;          /* ATT_TRANSFERRED */
    union
    {
        ATTTTransferredEvent_t    transferredEvent;
    } u;
} ATTEvent_t;

typedef struct ATTTTransferredEvent_t {
    ATTOOriginalCallInfo_t    originalCallInfo;
    CalledDeviceID_t          distributingDevice;
    ATTUCID_t                 ucid;
    ATTTTrunkList_t           trunkList;
    DeviceHistory_t           deviceHistory;
    CalledDeviceID_t          distributingVDN;
} ATTTTransferredEvent_t;

typedef struct ATTOOriginalCallInfo_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;
    unsigned char             flexibleBilling;
    DeviceHistory_t           deviceHistory;
} ATTOOriginalCallInfo_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_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_VECTURING_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 {
    unsigned short              count;
    short                       value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE      25

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 provided */
    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;

#define ATT_MAX_USER_INFO    129

typedef struct ATTUserToUserInfo_t {
    ATTUIProtocolType_t    type;
    struct
    {
        short                length;        /* 0
indicates no UII */
        unsigned char        value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUIProtocolType_t {
    UII_NONE = -1,                /* indicates not
specified */
    UII_USER_SPECIFIC = 0,        /* user-specific */
    UII_IA5_ASCII = 4            /* null-terminated
ASCII */
} ATTUIProtocolType_t;

typedef char    ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char    hasInfo;        /* if FALSE, no
call originator info */
    short            callOriginatorType;
} ATTCallOriginatorInfo_t;

typedef struct DeviceHistory_t {
    unsigned int        count;        /* at most 1 */
    DeviceHistoryEntry_t    *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t            olddeviceID;
    CSTAEventCause_t        cause;
    ConnectionID_t        oldconnectionID;
} DeviceHistoryEntry_t;

```

Private Data Version 6 Syntax

```

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

/* ATTV6TransferredEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV6_TRANSFERRED */
    union
    {
        ATTV6TransferredEvent_t    v6transferredEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV6TransferredEvent_t {
    ATTV6OriginalCallInfo_t    originalCallInfo;
    CalledDeviceID_t            distributingDevice;
    ATTUCID_t                    ucid;
    ATTTTrunkList_t              trunkList;
} ATTV6TransferredEvent_t;

typedef struct ATTV6OriginalCallInfo_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;
    unsigned char                flexibleBilling;
} ATTV6OriginalCallInfo_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_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 struct ATTUnicodeDeviceID_t {
    unsigned short      count;
    short               value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE      25

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 provided */
    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;

#define ATT_MAX_USER_INFO      129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;          /* 0
indicates no UUI */

```



```

        unsigned char                value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUIProtocolType_t {
    UUI_NONE = -1,                        /* indicates not
specified */
    UUI_USER_SPECIFIC = 0,                /* user-specific */
    UUI_IA5_ASCII = 4,                    /* null-terminated
ASCII */
} ATTUIProtocolType_t;

typedef char ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char    hasInfo;            /* if FALSE, no
callOriginatorType */
    short            callOriginatorType;
} ATTCallOriginatorInfo_t;

```

Private Data Version 5 Syntax

```

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

/* ATTV5TransferredEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV5_TRANSFERRED */
    union
    {
        ATTV5TransferredEvent_t    v5transferredEvent;
    } 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;
    unsigned char               flexibleBilling;
} ATTV5OriginalCallInfo_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_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 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 {
    unsigned short    count;
    short            value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE    25

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 provided */
    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 no UUI */
        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 {
    unsigned char    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_t    eventType;          /* ATTV4_TRANSFERRED */
    union
    {
        ATTV4TransferredEvent_t    v4transferredEvent;
    } 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 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 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 enum ATTUserEnteredCodeType_t {
    UE_NONE = -1,                   /* indicates not provided */
    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 no UUI */
        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 Versions 2 and 3 Syntax

```

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

/* ATTV3TransferredEvent - CSTA Unsolicited Event Private Data */

typedef struct
{
    ATTEventType_t      eventType;          /* ATTV3_TRANSFERRED */
    union
    {
        ATTV3TransferredEvent_t      v3transferredEvent;
    } u;
} ATTEvent_t;

typedef struct ATTV3TransferredEvent_t {
    ATTV4OriginalCallInfo_t      originalCallInfo;
} 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 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 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 enum ATTUserEnteredCodeType_t {
    UE_NONE = -1,                  /* indicates not provided */
    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 no UUI */
        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;
```

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 non-answering station.

- The Diverted Event Report is provided over `cstaMonitorDevice()` monitor requests when the call is redirected from a non-answering agent. This event is not provided if the call is queued again to the split or delivered to another agent in the split.
- The Queued Event Report will be generated if the call queues after being redirected.
- The Call Cleared Event Report – If the call cannot re-queue after the call has been redirected from the non-answering 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 queuing 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 Communication Manager call center configuration.
- A 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 for unsupervised transfers 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 TSAPI Service sends when the call answers), not the CSTA Delivered Event (which the TSAPI Service sends when the call rings). For an application to do an ANI screen pop on an unsupervised transfer, it must look in the proper CSTA Event.

If the design considerations in Chapter 3 are not followed, then ANI screen pop for consultation transfers 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 Transferred Event (which the TSAPI Service 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, non-split 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 associated `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 are 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 monitor 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.

Starting ASAI link ver



NOTE:

A principal station with bridged call appearances 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 the "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 be monitored using `cstaMonitorCallsViaDevice()`. Interactions between event reporting and call vectoring are shown in [Table 20](#).

Table 20: Interactions Between Feedback and Call Vectoring

Vector Step or Command	Event Report	When Sent	Cause
Vector Initialization	Delivered ¹² (to ACD device)	encountered	
Queue to Main	Queued	successfully queues	
	Failed	queue full, no agents logged in	queue full
Check Backup	Queued	successfully queues	
	Failed	queue full, no agents logged in	queue full
Messaging Split	Queued	successfully queues	
	Failed	queue full, no agents logged in	queue full
Announcement	none		
Wait	none		
GoTo	none		
Stop	none		

¹² Only reported for a VDN/ACD split monitored using `cstaMonitorCallsViaDevice()`.

Table 20: Interactions Between Feedback and Call Vectoring

Vector Step or Command	Event Report	When Sent	Cause
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	

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 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 Queuing

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

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 Button

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 associated with all calls received prior to the CTI link failure is no longer reliable. When a CTI link failure occurs, all call records are destroyed and information such as User To User Info and User Entered Codes are deleted from the TSAPI Service. If the link is restored in time, the call events may resume for new monitor requests (note that when CTI link is re-initialized, all monitor associations are aborted), but the Original Call Information for calls that existed 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

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 that 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 contain 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 Queuing

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 Queuing

`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 or service 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 as well as to all other monitor requests for that call.

Temporary Bridged Appearances

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 (non-answering 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()` requests. 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

The Routing Service Group provides 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 and events are provided:

- [Route End Event](#) on page 893
- [Route End Service \(TSAPI Version 2\)](#) on page 898
- [Route End Service \(TSAPI Version 1\)](#) on page 901
- [Route Register Abort Event](#) on page 903
- [Route Register Cancel Service](#) on page 905
- [Route Register Service](#) on page 908
- [Route Request Event \(TSAPI Version 2\)](#) on page 911
- [Route Request Event \(TSAPI Version 1\)](#) on page 932
- [Route Select Service \(TSAPI Version 2\)](#) on page 936
- [Route Select Service \(TSAPI Version 1\)](#) on page 952
- [Route Used Event \(TSAPI Version 2\)](#) on page 954
- [Route Used Event \(TSAPI Version 1\)](#) on page 958

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:

<code>routeRegisterReqID</code>	[mandatory] Contains the handle to the routing registration session for which the application is providing routing services. The application received this handle in a <code>CSTARouteRegisterReqConf-Event</code> confirmation to a <code>cstaRouteRegisterReq()</code> request.
<code>routingCrossRefID</code>	[mandatory] Contains the handle to the CSTA call routing dialog for a call. The application previously received this handle in the <code>CSTARouteRequestExtEvent</code> for the call. This is the routing dialog that the switch is ending.
<code>errorValue</code>	[mandatory] Contains the cause code for the reason why the switch is ending the routing dialog. One of the following values will be returned: • <code>GENERIC_UNSPECIFIED (0) (CS0/16)</code> – The call has been routed successfully. – The adjunct route request to route using Network Call Redirection (NCR) resulted in the call not being routed by NCR because of an internal system error. • <code>GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (CS0/50)</code> 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). • <code>INVALID_CALLING_DEVICE (5) (CS3/15)</code> 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 the 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 Network Call Transfer (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 cannot be set up.
 - The adjunct route request to route using NCR resulted in the call not being routed by NCR because of a PSTN Network Call Deflection (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 `CSTARouteRequestExtEvent` (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 the following call scenario:

- The TSAPI Service sends a Route Request to the application on behalf of the switch.
- The caller drops the call.
- The application sends a Route Select Request to the TSAPI Service.
- The TSAPI Service sends a Route End Event (`errorValue = NO_ACTIVE_CALL`) to the application before receiving the Route Select Request.
- The TSAPI Service receives the Route Select Request, but the call has been dropped.

- The TSAPI Service sends a Universal Failure for the Route Select request (`errorValue = INVALID_CROSS_REF_ID`) to the application.

Syntax

```

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

/* CSTARouteEndEvent - Route Select Service Response */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAEVENTREPORT */
    EventType_t      eventType;          /* CSTA_ROUTE_END */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            union
            {
                CSTARouteEndEvent_t      routeEnd;
            } u;
        } cstaEventReport;
    } event;
} CSTAEvent_t

typedef struct CSTARouteEndEvent_t {
    RouteRegisterReqID_t      routeRegisterReqID,
    RoutingCrossRefID_t       routingCrossRefID,
    CSTAUniversalFailure_t    errorValue,
} CSTARouteEndEvent_t;

typedef long    RouteRegisterReqID_t;
typedef long    RoutingCrossRefID_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:

<code>routeRegisterReqID</code>	[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 <code>CSTARouteRegisterReqConfEvent</code> confirmation to a <code>cstaRouteRegisterReq()</code> request.
<code>routingCrossRefID</code>	[mandatory] Contains the handle to the CSTA call routing dialog for a call. The routing server application previously received this handle in the <code>CSTARouteRequestExtEvent</code> for the call. This is the routing dialog that the application is terminating.
<code>errorValue</code>	[mandatory] Contains the cause code for the reason why the application is terminating the routing dialog. Any CSTA <code>universalFailure</code> error code can be sent.

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, the 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 `cstaRouteEndInv()` request.
- `RESOURCE_BUSY (33)` The routing server is too busy to handle the route request.

- `RESOURCE_OUT_OF_SERVICE` (34) The routing service temporarily unavailable due to internal problem (for example, the database is out of service).

Ack Parameters:

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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors - universalFailure](#) on page 997.

- `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()` service request, 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:
 - The TSAPI Service sends a `CSTARouteRequestEvent` to the application on behalf of the switch.
 - The caller drops the call.
 - The application sends a `cstaRouteEndInv()` request to the TSAPI Service.
 - The TSAPI Service sends a `CSTARouteEndEvent` (`errorValue = NO_ACTIVE_CALL`) to the application before receiving the Route Select Request.
 - The TSAPI Service receives the `cstaRouteEndInv()` request, but the call has been dropped.
 - The TSAPI Service sends a `universalFailure` for the `cstaRouteEndInv()` request (`errorValue = INVALID_CROSS_REF_ID`) to the 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:

- The TSAPI Service sends a `CSTARouteRequestEvent` to the application on behalf of the switch.
- The caller drops the call.
- The application sends a `cstaRouteSelect()` to the TSAPI Service.
- The TSAPI Service sends a `CSTARouteEndEvent` (`errorValue = NO_ACTIVE_CALL`) to the application before receiving the Route Select request.
- The TSAPI Service receives the `cstaRouteSelect()` request, but the call has been dropped.
- The TSAPI Service sends a `CSTARouteEndEvent` (`errorValue = INVALID_CROSS_REF_ID`) to the 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:

<code>routeRegisterReqID</code>	[mandatory] Contains the handle to the routing registration session for which the application is providing routing services. The application received this handle in a <code>CSTARouteRegisterReqConfEvent</code> confirmation to a <code>cstaRouteRegisterReq()</code> 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, then 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 comes back 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_t     eventClass;          /* CSTAEVENTREPORT */
    EventType_t      eventType;          /*
CSTA_ROUTE_REGISTER_ABORT */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            union
            {
                CSTARouteRegisterAbortEvent_t  registerAbort;
            } 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 `CSTARouteRequestEvent(s)` for the specific routing device associated with the `routeRegisterReqID` to the requesting client application. The TSAPI Service will send any further `CSTARouteRequestEvent(s)` from the routing device to the default routing server application, if there is one registered.

Service Parameters:

<code>routeRegisterReqID</code>	[mandatory] Contains the handle to the routing registration session for which the application is canceling. The routing server application received this handle in a <code>CSTARouteRegisterReqConfEvent</code> confirmation to a <code>cstaRouteRegisterReq()</code> request.

Ack Parameters:

None for this service.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The error parameter in this event may contain the following error value, or one of the error values described in Table 21: Common switch-related CSTA Service errors - universalFailure on page 997. • <code>INVALID_CSTA_DEVICE_IDENTIFIER (12)</code> – An invalid <code>routeRegisterReqID</code> 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 time 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_t     eventClass;          /* CSTACONFIRMATION */
    EventType_t      eventType;          /*
CSTA_ROUTE_REGISTER_CANCEL_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTARouteRegisterCancelConfEvent_t
            }
        } routeCancel;
    } u;
    } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTARouteRegisterCancelConfEvent_t {
    RouteRegisterReqID_t      routeRegisterReqID;
} CSTARouteRegisterCancelConfEvent_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 a specific device. As such, the application will receive `CSTARouteRequestExtEvent(s)` for that 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 than 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 an application to re-register as a routing device. For example, if the application loses its connection to the AE Services server and then establishes a new connection to the AE Services server (i.e., opens a new ACS stream), the application is allowed to re-register itself as a routing server for the same device as long as the following criteria are met:

- The login (`LoginID_t`) provided by the application in the `acsOpenStream()` request matches that of the previously registered application
- The application name (`AppName_t`) provided by the application in the `acsOpenStream()` request matches that of the previously registered application
- The IP address of the client machine matches that of the previously registered application.

Service Parameters:

`routingDevice` [mandatory] Contains the device identifier of the routing device for which the application is registering to be the routing server. A valid routing device on Communication Manager 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 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 value, or one of the error values described in [Table 21: Common switch-related CSTA Service errors - - universalFailure](#) on page 997.

- `OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44)` – The specified routing device already has a registered routing server.

Detailed Information:

- The `cstaRouteRegisterReq()` service 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; /* CSTACONFIRMATION */
    EventType_t      eventType; /* CSTA_ROUTE_REGISTER_REQ_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTARouteRegisterReqConfEvent_t
            }
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTARouteRegisterReqConfEvent_t {
    RouteRegisterReqID_t      registerReqID;
} CSTARouteRegisterReqConfEvent_t;

```

Route Request Event (TSAPI Version 2)

Summary

- **Direction:** Switch to Client
- **Event:** `CSTARouteRequestExtEvent`
- **Private Data Event:** `ATTRouteRequestEvent`(private data 15 and later) ,
`ATTV14RouteRequestEvent` (private data version 14),
`ATTV13RouteRequestEvent` (private data version 7 and later),
`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`, `data26`, `data38`

Functional Description:

The TSAPI Service sends a `CSTARouteRequestExtEvent` to a routing server application in order to request a destination for a call arrived on a routing device. 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:

<code>routeRegisterReqID</code>	[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 <code>CSTARouteRegisterReqConfEvent</code> confirmation to a <code>cstaRouteRegisterReq()</code> request.
<code>routingCrossRefID</code>	[mandatory] Contains the handle for the routing dialog of this call. This identifier is unique within a routing session identified by the <code>routeRegisterReqID</code> .
<code>currentRoute</code>	[mandatory] Specifies the destination of the call. This is the VDN extension number first entered by the call (see Detailed Information)
<code>callingDevice</code>	[optional – supported] Specifies 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.	
<code>routedCall</code>	[optional – supported] Specifies the <code>callID</code> of the call that is to be routed. This is the <code>connectionID</code> of the routed call at the routing device.
<code>routedSelAlgorithm</code>	[optional – partially supported] Indicates the type of routing algorithm requested. It is set to <code>SV_NORMAL</code> .
<code>priority</code>	[optional – not supported] Indicates the priority of the call and may affect selection of alternative routes.
<code>setupInformation</code>	[optional – not supported] Contains an ISDN call setup message if available.

Private Parameters:

<code>trunkGroup</code>	[optional] Specifies the trunk group number from which the call is originated. The <code>callingDevice</code> and trunk parameters are mutually exclusive. This parameter is supported by private data version 5 and later only.
<code>trunkMember</code>	[optional] This parameter specifies the trunk member number from which this call originated.
<code>trunk</code>	[optional] Specifies the trunk group number from which the call is originated. This parameter is supported by private data versions 2, 3, and 4.
<code>lookaheadInfo</code>	[optional] Specifies the lookahead interflow information received from the incoming call that is to be routed. The 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. The routing server application may use the lookahead interflow information to determine the destination of the call. Please refer to the Communication Manager 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.
<code>userEnteredCode</code>	[optional] Specifies the code/digits that may have been entered by the caller through the Communication Manager call prompting feature or the collected digits feature. If the <code>userEnteredCode</code> code is set to "UE_NONE", no <code>userEnteredCode</code> private data is provided with this event.
<code>userInfo</code>	[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) bytes, with a call. It may be a customer number, credit card number, alphanumeric digits, or a binary string.

➤ NOTE:

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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.

<code>ucid</code>	[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 <code>ucid</code> contains the <code>ATT_NULL_UCID</code> (a 20-character string of all zeros). This parameter is supported by private data version 5 and later only.
<code>callOriginator</code>	[optional] Specifies the <code>callOriginatorInfo</code> of the call originator such as coin call, 800-service call, or cellular call. This information is from the network, not from Communication Manager. 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 provided in the Detailed Information sub-section of the "Delivered Event" section. This parameter is supported by private data version 5 and later only.
<code>flexibleBilling</code>	[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 <code>TRUE</code> , 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.
<code>deviceHistory</code>	The <code>deviceHistory</code> parameter type specifies a list of <code>DeviceIDs</code> 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 <code>deviceHistory</code> 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).

Conceptually, the `deviceHistory` parameter consists of a list of entries, where each entry contains information about a `DeviceID` that had previously been associated with the call, and the list is ordered from the first device that left the call to the device that most recently left the call. However, for AE Services, the list will contain at most one entry.

The entry consists of:

- `olddeviceID` – 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 may also be:

- “Not Known” – indicates that the device identifier cannot be provided.
- “Not Required” – indicates that there are no devices that have left the call. If this value is provided, the cause and `oldconnectionID` are not provided.
- `cause` – the reason the device left the call or was redirected. This information should be consistent with the `cause` provided in the event that represented the device leaving the call (for example, the cause provided in the Diverted, Transferred, or Connection Cleared event).
- `oldconnectionID` – 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:

The Device History cannot be guaranteed for events that happened before monitoring started. Notice that the cause value should be `EC_NETWORK_SIGNAL` 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.

NOTE:

`data26` and `data38` are use for Avaya internal

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 number through which the switch will send the Route Request to the TSAPI Service.
- Multiple adjunct routing commands are allowed in a call vector. 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, 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()` service request, the switch continues vector processing.
- A `CSTARouteRequestExtEvent` request will not affect the Call Event Reports.
- Like the Delivered and Established Events, the Route Request `currentRoute` parameter contains the called device. 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 latter 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 Request */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAREQUEST */
    EventType_t      eventType;          /* CSTA_ROUTE_REQUEST_EXT */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTARouteRequestExtEvent_t  routeRequestExt;
            } u;
        } cstaRequest;
    } event;
} CSTAEvent_t;

typedef struct CSTARouteRequestExtEvent_t {
    RouteRegisterReqID_t      routeRegisterReqID;
    RoutingCrossRefID_t      routingCrossRefID;
    CalledDeviceID_t          currentRoute;
    CallingDeviceID_t         callingDevice;
    ConnectionID_t            routedCall;
    SelectValue_t             routedSelAlgorithm;
    unsigned char             priority;
    SetUpValues_t             setupInformation;
} CSTARouteRequestExtEvent_t;

typedef long      RouteRegisterReqID_t;
typedef long      RoutingCrossRefID_t;

typedef ExtendedDeviceID_t      CallingDeviceID_t;
typedef ExtendedDeviceID_t      CalledDeviceID_t;

typedef struct ExtendedDeviceID_t {
    DeviceID_t      deviceID;
    DeviceIDType_t  deviceIDType;
}

```

```

        DeviceIDStatus_t      deviceIDStatus;
    } ExtendedDeviceID_t;

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

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

typedef enum SelectValue_t {
    SV_NORMAL = 0,
    SV_LEAST_COST = 1,
    SV_EMERGENCY = 2,
    SV_ACD = 3,
    SV_USER_DEFINED = 4
} SelectValue_t;

typedef struct SetupValues_t {
    unsigned int      length;
    unsigned char     *value;
} SetupValues_t;

```

Private Data 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.

Private Data Version 15 and Later Syntax

The `deviceHistory` parameter is added for private data version 7.

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

/* ATTRouteRequestEvent - CSTA Request Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_ROUTE_REQUEST */
    union
    {
        ATTRouteRequestEvent_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;
    unsigned char              flexibleBilling;
    DeviceID_t                 trunkMember;
    DeviceHistory_t            deviceHistory;
    Reserved16_t               data26;
    Reserved31_t               data38;
} ATTRouteRequestEvent_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_VECTORIZING_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 {
    unsigned short      count;
    short               value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE      25

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 provided */
    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;

```

```

#define ATT_MAX_USER_INFO      129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t      type;
    struct
    {
        short                  length;          /* 0
indicates no UUI */
        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 {
    unsigned char    hasInfo;      /* if FALSE, no
                                     *
callOriginatorType */
    short            callOriginatorType;
} ATTCallOriginatorInfo_t;

typedef struct DeviceHistory_t {
    unsigned int      count;        /* at most 1 */
    DeviceHistoryEntry_t *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t          olddeviceID;
    CSTAEventCause_t     cause;
    ConnectionID_t       oldconnectionID;
} DeviceHistoryEntry_t;

```

Private Data Version 7 – 13 Syntax

The `deviceHistory` parameter is added for private data version 7.

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

/* ATTV13RouteRequestEvent - CSTA Request Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_ROUTE_REQUEST */
    union
    {
        ATTV14RouteRequestEvent_t    v6routeRequest;
    } u;
} ATTEvent_t;

typedef struct ATTV13RouteRequestEvent_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;
} ATTV13RouteRequestEvent_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 struct ATTUnicodeDeviceID_t {
    unsigned short    count;
    short            value[64];
} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE    25

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 provided */
    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;

#define ATT_MAX_USER_INFO    129

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t    type;
    struct
    {
        short                length;    /* 0
indicates no UUI */

```

```

        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 */
} ATTUUIProtocolType_t;

typedef char ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char    hasInfo;          /* if FALSE, no
callOriginatorType */
    short            callOriginatorType;
} ATTCallOriginatorInfo_t;

typedef struct DeviceHistory_t {
    unsigned int      count;           /* at most 1 */
    DeviceHistoryEntry_t *deviceHistoryList;
} DeviceHistory_t;

typedef struct DeviceHistoryEntry_t {
    DeviceID_t        olddeviceID;
    CSTAEventCause_t  cause;
    ConnectionID_t    oldconnectionID;
} DeviceHistoryEntry_t;

```

Private Data Version 6 Syntax

```

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

/* ATTV6RouteRequestEvent - CSTA Request Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV6_ROUTE_REQUEST */
    union
    {
        ATTV6RouteRequestEvent_t    v6routeRequest;
    } u;
} ATTEvent_t;

typedef struct ATTV6RouteRequestEvent_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;
} ATTV6RouteRequestEvent_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 struct ATTUnicodeDeviceID_t {
    unsigned short            count;

```

```

        short                                value[64];
    } ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE    25

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 provided */
    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;

#define ATT_MAX_USER_INFO    129

typedef struct ATTUserToUserInfo_t {
    ATTUIProtocolType_t        type;
    struct
    {
        short                                length;        /* 0
indicates no UII */
        unsigned char                        value[ATT_MAX_USER_INFO];
    } data;
} ATTUserToUserInfo_t;

typedef enum ATTUIProtocolType_t {
    UII_NONE = -1,                                /* indicates not
specified */
    UII_USER_SPECIFIC = 0,                        /* user-specific */
    UII_IA5_ASCII = 4                             /* null-terminated
ASCII
*
character string */
} ATTUIProtocolType_t;

typedef char    ATTUCID_t[64];

typedef struct ATTCallOriginatorInfo_t {
    unsigned char    hasInfo;        /* if FALSE, no
*
callOriginatorType */

```

```
        short                callOriginatorType;  
    } ATTCallOriginatorInfo_t;
```

Private Data Version 5 Syntax

```

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

/* ATTV5RouteRequestEvent - CSTA Request Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV5_ROUTE_REQUEST */
    union
    {
        ATTV5RouteRequestEvent_t    v5routeRequest;
    } u;
} ATTEvent_t;

typedef struct ATTV5RouteRequestEvent_t {
    DeviceID_t                trunkGroup;
    ATTLookaheadInfo_t        lookaheadInfo;
    ATTUserEnteredCode_t      userEnteredCode;
    ATTV5UserToUserInfo_t     userInfo;
    ATTUCID_t                 ucid;
    ATTCallOriginatorInfo_t   callOriginatorInfo;
    unsigned char             flexibleBilling;
} ATTV5RouteRequestEvent_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 struct ATTUnicodeDeviceID_t {
    unsigned short            count;
    short                     value[64];

```

```

} ATTUnicodeDeviceID_t;

#define ATT_MAX_USER_CODE      25

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 provided */
    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 no UUI */
        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 {
    unsigned char    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 Request Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV4_ROUTE_REQUEST */
    union
    {
        ATTV4RouteRequestEvent_t    v4routeRequest;
    } u;
} ATTEvent_t;

typedef struct ATTV4RouteRequestEvent_t {
    DeviceID_t                trunk;
    ATTV4LookaheadInfo_t      lookaheadInfo
    ATTUserEnteredCode_t      userEnteredCode;
    ATTV5UserToUserInfo_t     userInfo;
} ATTV4RouteRequestEvent_t;

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_VECTORIZING_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;

#define ATT_MAX_USER_CODE      25

```



```

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 provided */
    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 no UUI */
        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;

```

Route Request Event (TSAPI Version 1)

Summary

- **Direction:** Switch to Client
- **Event:** `CSTARouteRequestEvent`
- **Service Parameters:** `routeRegisterReqID`, `routingCrossRefID`, `currentRoute`, `callingDevice`, `routedCall`, `routedSelAlgorithm`, `priority`, `setupInformation`

Functional Description:

The TSAPI Service sends a `CSTARouteRequestEvent` to a routing server application in order to request a destination for a call arrived on a routing device. 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 `CSTARouteRequestEvent` via a `cstaRouteSelect()` request that specifies a destination for the call or a `cstaRouteEnd()` request, if the application has no destination for the call.

Service Parameters:

<code>routeRegisterReqID</code>	[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 <code>CSTARouteRegisterReqConfEvent</code> confirmation to a <code>cstaRouteRegisterReq()</code> request.
<code>routingCrossRefID</code>	[mandatory] Contains the handle for the routing dialog of this call. This identifier is unique within a routing session identified by the <code>routeRegisterReqID</code> .
<code>currentRoute</code>	[mandatory] Specifies the destination of the call. This is the VDN extension number first entered by the call (see Detailed Information)
<code>callingDevice</code>	[optional – supported] Specifies 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.	
<code>routedCall</code>	[optional – supported] Specifies the <code>callID</code> of the call that is to be routed. This is the <code>connectionID</code> of the routed call at the routing device.
<code>routedSelAlgorithm</code>	[optional – partially supported] Indicates the type of routing algorithm requested. It is set to <code>SV_NORMAL</code> .
<code>priority</code>	[optional – not supported] Indicates the priority of the call and may affect selection of alternative routes.
<code>setupInformation</code>	[optional – not supported] Contains an ISDN call setup message if available.

Detailed Information:

- 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 `CSTARouteEndEvents`.
- If application terminates the `CSTARouteRequestEvent` request via a `cstaRouteEnd()` service request, the switch continues vector processing.
- A `CSTARouteRequestEvent` request will not affect the Call Event Reports.

Syntax

```

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

/* CSTARouteRequestEvent - CSTA Request */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;      /* CSTAREQUEST */
    EventType_t      eventType;      /* CSTA_ROUTE_REQUEST */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;      /* Not used for this event
*/
            union
            {
                CSTARouteRequestEvent_t routeRequest;
            } u;
        } cstaRequest;
    } event;
} CSTAEvent_t;

typedef struct CSTARouteRequestEvent_t {
    RouteRegisterReqID_t  routeRegisterReqID;
    RoutingCrossRefID_t   routingCrossRefID;
    DeviceID_t            currentRoute;
    DeviceID_t            callingDevice;
    ConnectionID_t        routedCall;
    SelectValue_t         routedSelAlgorithm;
    unsigned char          priority;
    SetUpValues_t         setupInformation;
} CSTARouteRequestEvent_t;

typedef long            RouteRegisterReqID_t;
typedef long            RoutingCrossRefID_t;
typedef char            DeviceID_t[64];

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

```

```
} ConnectionID_t;

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

typedef enum SelectValue_t {
    SV_NORMAL = 0,
    SV_LEAST_COST = 1,
    SV_EMERGENCY = 2,
    SV_ACD = 3,
    SV_USER_DEFINED = 4
} SelectValue_t;

typedef struct SetupValues_t {
    unsigned int      length;
    unsigned char     *value;
} SetupValues_t;
```

Route Select Service (TSAPI Version 2)

Summary

- Direction: Client to Switch
- Function: `cstaRouteSelectInv()`
- Private Data Function: `attv15RouteSelect()` (private data version 15 and later), `attV13RouteSelect()` (private data version 7-13), `attV6RouteSelect()` (private data version 6), `attV5RouteSelect()` (private data versions 2, 3, 4, and 5)
- Service Parameters: `routeRegisterReqID`, `routingCrossRefID`, `routeSelected`, `remainRetry`, `setupInformation`, `routeUsedReq`
- Private Parameters: `callingDevice`, `directAgentCallSplit`, `priorityCalling`, `destRoute`, `collectCode`, `userProvidedCode`, `userInfo`, `redirectType`, `data17`, `data27`, `data38`
- 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:

<code>routeRegisterReqID</code>	[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 <code>CSTARouteRegisterReqConfEvent</code> confirmation to a <code>cstaRouteRegisterReq()</code> request.
<code>routingCrossRefID</code>	[mandatory] Contains the handle for the routing dialog of this call. The application previously received this handle in the <code>CSTARouteRequestExtEvent</code> for the call.
<code>routeSelected</code>	[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).
<code>remainRetry</code>	[optional – not supported] Specifies the number of times that the application is willing to receive a <code>CSTARouteRequestExtEvent</code> for this call in case the switch needs to request an alternate route.
<code>setupInformation</code>	[optional – not supported] Contains a revised ISDN call setup message that the switch will use to route the call.
<code>routeUsedReq</code>	[optional – supported] Indicates a request to receive a <code>CSTARouteUsedExtEvent</code> for the call. • If specified, the TSAPI Service always returns the same destination information that is specified in the <code>routeSelected</code> and <code>destRoute</code> of this <code>cstaRouteSelectInv()</code>.

Private Parameters:

<code>callingDevice</code>	[optional] Specifies the calling device. A <code>NULL</code> specifies that this parameter is not present.
<code>directAgentCallSplit</code>	[optional] Specifies the ACD agent's split extension for 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 by <code>routeSelected</code> must be logged into this split. A <code>NULL</code> parameter specifies that this is not a Direct-Agent call.
<code>priorityCalling</code>	[mandatory] Specifies the priority of the call. Values are "On" (<code>TRUE</code>) or "Off" (<code>FALSE</code>). When "On" is selected, a priority call is placed if the <code>routeSelected</code> is an on-PBX destination. When "On" is selected for an off-PBX destination, the call will be denied.
<code>destRoute</code>	[optional] Specifies the TAC/ARS/AAR information for off-PBX destinations, if the information is not included in the <code>routeSelected</code> . A <code>NULL</code> parameter specifies no TAC/ARS/AAR information.
<code>collectCode</code>	[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 not necessarily collected while the call is 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 an Entered Digits event report.

Only incoming trunk calls are eligible for this feature. Further, Disconnect Supervision for incoming calls must be enabled for the trunk group.

The feature "Vectoring (Prompting)" must be enabled on Avaya Communication Manager for this feature to work.

- To use this feature, the `type` parameter in `collectCode` must be set to `UC_TONE_DETECTOR`. Otherwise, the `type` parameter should be set to `UC_NONE`.
- The `digitsToBeCollected` parameter in `collectCode` indicates the total number of digits to collect (1-24).
- The `timeout` parameter in `collectCode` indicates how many seconds (1-31) the tone detector will continue to collect digits after the first digit is received.

- The `collectParty` parameter in `collectCode` indicates to which party on the call the tone detector should listen. The value of this parameter is ignored. Currently, the call originator is the only option supported.
- The `specificEvent` parameter in `collectCode` indicates when the tone detector should be released. The only supported value is `SE_ANSWER` (far end answer/connect).
- 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.
- 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.

- To use this feature, the `type` parameter in `userProvidedCode` must be set to `UP_DATA_BASE_PROVIDED`. Otherwise, the `type` parameter should be set to `UP_NONE`.
- The `data` parameter in `userProvidedCode` indicates which digits (0-9, *, and #) to send. Up to 24 digits may be sent.

- 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.

`userInfo`

[optional] Contains user-to-user information. This parameter allows the application to associate caller information, up to 32 (private data versions 2-5) or 96 (private data versions 6 and later) 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.

An application using private data version 5 or earlier can only receive a maximum of 32 bytes of data in `userInfo`, regardless of the size of the data 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 Network Call Redirection (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:

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 one of the following error values, or one of the error values described in [Table 21: Common switch-related CSTA Service errors -- universalFailure](#) on page 997.

- `INVALID_CSTA_DEVICE_IDENTIFIER` (12) – An invalid `routeRegisterReqID` has been specified in the `cstaRouteSelectInv()` 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 the following call scenario:

- The TSAPI Service sends a `CSTARouteRequestExtEvent` to the application on behalf of the switch.
- The caller drops the call.
- The application sends a `cstaRouteSelectInv()` request to the TSAPI Service.
- The TSAPI Service sends a `CSTARouteEndEvent` (`errorValue = NO_ACTIVE_CALL`) to the application before receiving the Route Select request.
- The TSAPI Service receives the `cstaRouteSelectInv()` request, but the call has been dropped.
- The TSAPI Service sends a `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,
    RouteRegisterReqID_t routeRegisterReqID,
    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 {
    unsigned int    length;
    unsigned char   *value;
} SetUpValues_t;
```

Private Data Version 15 and Later Syntax

```

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

/*
 * attv15RouteSelect() - Service Request Private Data
 * Formatting Function
 */
RetCode_t attv15RouteSelect(
    ATTPrivateData_t      *attPrivateData,
    DeviceID_t            *callingDevice,
    DeviceID_t            *directAgentCallSplit,
    Boolean                priorityCalling,
    DeviceID_t            *destRoute,
    ATTUserCollectCode_t   *collectCode,
    ATTUserProvidedCode_t  *userProvidedCode,
    ATTUserToUserInfo_t    *userInfo,
    ATTRedirectType_t      redirectType,
    DeviceID_t            *data17,
    Reserved7_t            data27,
    Reserved31_t           *data38);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTUserCollectCode_t {
    ATTCollectCodeType_t  type;
    short                digitsToBeCollected; /*
1-24 digits */
    short                timeout;
/* 0-63 secs */
    ConnectionID_t       collectParty; /*
not supported

* (defaults to
* call
* originator) */
    ATTSpecificEvent_t   specificEvent; /* not
supported

* (defaults to

```

```

        * 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;

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

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct {
        short length; /* 0
indicates no UUI */
        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 ATTRedirectType_t {
    VDN = 0,
    NETWORK = 1

```

```
} ATTRedirectType_t;
```

Private Data Version 7 and Later Syntax

```

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

/*
 * attV13RouteSelect() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV13RouteSelect(
    ATTPrivateData_t      *attPrivateData,
    DeviceID_t            *callingDevice,
    DeviceID_t            *directAgentCallSplit,
    Boolean               priorityCalling,
    DeviceID_t            *destRoute,
    ATTUserCollectCode_t  *collectCode,
    ATTUserProvidedCode_t *userProvidedCode,
    ATTUserToUserInfo_t   *userInfo,
    ATTRedirectType_t     redirectType);

typedef struct ATTPrivateData_t {
    char            vendor[32];
    unsigned short  length;
    char            data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTUserCollectCode_t {
    ATTCollectCodeType_t  type;
    short                digitsToBeCollected; /*
1-24 digits */
    short                timeout;
/* 0-63 secs */
    ConnectionID_t       collectParty; /*
not supported

    * (defaults to

    * call

    * originator) */
    ATTSpecificEvent_t   specificEvent; /* not
supported

    * (defaults to

    * 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;

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

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t type;
    struct {
        short length; /* 0
indicates no UUI */
        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 ATTRedirectType_t {
    VDN = 0,
    NETWORK = 1
} ATTRedirectType_t;

```

Private Data Version 6 Syntax

```

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

/*
 * attV6RouteSelect() - Service Request Private Data
 * Formatting Function
 */

RetCode_t attV6RouteSelect(
    ATTPrivateData_t      *attPrivateData,
    DeviceID_t            *callingDevice,
    DeviceID_t            *directAgentCallSplit,
    Boolean               priorityCalling,
    DeviceID_t            *destRoute,
    ATTUserCollectCode_t  *collectCode,
    ATTUserProvidedCode_t *userProvidedCode,
    ATTUserToUserInfo_t   *userInfo);

typedef struct ATTPrivateData_t {
    char            vendor[32];
    unsigned short  length;
    char            data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTUserCollectCode_t {
    ATTCollectCodeType_t  type;
    short                digitsToBeCollected; /*
1-24 digits */
    short                timeout; /*
/* 0-63 secs */
    ConnectionID_t       collectParty; /*
not supported

                                * (defaults to
                                * call
                                * originator) */
    ATTSpecificEvent_t    specificEvent; /* not
supported

                                * (defaults to
                                * 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;

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

typedef struct ATTUserToUserInfo_t {
    ATTUUIProtocolType_t    type;
    struct
    {
        short                length;            /* 0
indicates no UUI */
        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
 * Formatting Function
 */

RetCode_t attRouteSelect(
    ATTPrivateData_t      *attPrivateData,
    DeviceID_t            *callingDevice,
    DeviceID_t            *directAgentCallSplit,
    Boolean               priorityCalling,
    DeviceID_t            *destRoute,
    ATTUserCollectCode_t  *collectCode,
    ATTUserProvidedCode_t *userProvidedCode,
    ATTV5UserToUserInfo_t *userInfo);

typedef struct ATTPrivateData_t {
    char            vendor[32];
    unsigned short  length;
    char            data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

typedef struct ATTUserCollectCode_t {
    ATTCollectCodeType_t  type;
    short                digitsToBeCollected; /*
1-24 digits */
    short                timeout;
/* 0-63 secs */
    ConnectionID_t      collectParty; /*
not supported

    * (defaults to

    * call

    * originator) */
    ATTSpecificEvent_t  specificEvent; /* not
supported

    * (defaults to

    * 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;

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 struct ATTV5UserToUserInfo_t {
    ATTUUIProtocolType_t    type;
    struct
    {
        short                length;            /* 0
indicates no UUI */
        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;

```

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 the following call scenario:

- The TSAPI Service sends a `CSTARouteRequestEvent` to the application on behalf of the switch.
- The caller drops the call.
- The application sends a `cstaRouteSelect()` request to the TSAPI Service.
- The TSAPI Service sends a `CSTARouteEndEvent` (`errorValue = NO_ACTIVE_CALL`) to the application before receiving the Route Select request.
- The TSAPI Service receives the `cstaRouteSelect()` request, but the call has been dropped.
- The TSAPI Service sends a `CSTARouteEndEvent` for the `cstaRouteSelect()` request (`errorValue = INVALID_CROSS_REF_ID`) to the application.

Syntax

```

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

/* cstaRouteSelect() - Service Request */

RetCode_t    cstaRouteSelect(
    ACSHandle_t      acsHandle,
    RouteRegisterReqID_t  routeRegisterReqID,
    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 {
    unsigned 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 Route Select service request containing a destination. The `routeUsed` parameter and `destRoute` private parameter contain the same information specified in the `routeSelected` and `destRoute` parameters, respectively, of the previous `cstaRouteSelectInv()` request of this call. The `callingDevice` parameter contains the same calling device number provided in the previous `CSTARouteRequestExtEvent` of this call.

Service Parameters:

<code>routeRegisterReqID</code>	[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 <code>CSTARouteRegisterReqConfEvent</code> confirmation to a <code>cstaRouteRegisterReq()</code> request.
<code>routingCrossRefID</code>	[mandatory] Contains the handle for the routing dialog of this call. The application previously received this handle in the <code>CSTARouteRequestExtEvent</code> for the call.
<code>routeUsed</code>	[mandatory] Specifies the destination of the call. This parameter has the same destination specified in the <code>routeSelected</code> parameter of the previous <code>cstaRouteSelectInv()</code> request for this call.
<code>callingDevice</code>	[optional – supported] Specifies the call origination device. It contains the same calling device number provided in the previous <code>CSTARouteRequestExtEvent</code> for this call.

`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 for this call.

Detailed Information:

- Note that the number provided in the `routeUsed` parameter is from the `route-Selected` parameter of the previous `cstaRouteSelectInv()` request received by the TSAPI Service for this call. The 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 `dest-Route` parameter of the previous `cstaRouteSelectInv()` request received by the TSAPI Service for this call. The 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 `calling-Device` parameter of the previous `CSTARouteRequestExtEvent` sent by the TSAPI Service for this call.

Syntax

```

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

/* CSTARouteUsedExtEvent - Route Select Service Response */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAEVENTREPORT */
    EventType_t      eventType;          /* CSTA_ROUTE_USED_EXT */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            union
            {
                CSTARouteUsedExtEvent_t routeUsedExt;
            } u;
        } cstaEventReport;
    } event;
} CSTAEvent_t;

typedef struct CSTARouteUsedExtEvent_t {
    RouteRegisterReqID_t routeRegisterReqID;
    RoutingCrossRefID_t routingCrossRefID;
    CalledDeviceID_t routeUsed;
    CallingDeviceID_t callingDevice;
    unsigned char domain;
} CSTARouteUsedExtEvent_t;

```

Private Data 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
{
    ATTEventType_t    eventType;          /* ATT_ROUTE_USED */
    union
    {
        ATTRouteUsedEvent_t    routeUsed;
    }u;
} ATTEvent_t;

typedef struct ATTRouteUsedEvent_t {
    DeviceID_t    destRoute;
} ATTRouteUsedEvent_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 `CSTARouteUsedEvent` to provide a destination to the routing server application with the actual destination of a call for which the application previously sent a Route Select service request containing a destination. The `routeUsed` parameter contains the same information specified in the `routeSelected` parameter of the previous `cstaRouteSelect()` request for this call. The `callingDevice` parameter contains the same calling device number provided in the previous `CSTARouteRequestEvent` for this call.

Service Parameters:

<code>routeRegisterReqID</code>	[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 <code>CSTARouteRegisterReqConfEvent</code> confirmation to a <code>cstaRouteRegisterReq()</code> request.
<code>routingCrossRefID</code>	[mandatory] Contains the handle for the routing dialog of this call. The application previously received this handle in the <code>CSTARouteRequestExtEvent</code> for the call.
<code>routeUsed</code>	[mandatory] Specifies the destination of the call. This parameter has the same destination specified in the <code>routeSelected</code> parameter of the previous <code>cstaRouteSelect()</code> request for this call.
<code>callingDevice</code>	[optional – supported] Specifies the call origination device. It contains the same calling device number provided in the previous <code>CSTARouteRequestExtEvent</code> for this call.
<code>domain</code>	[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 <code>FALSE</code> . This does not mean that the call has (or has not) left Communication Manager. An application should ignore this parameter.

Detailed Information:

- The number provided in the `routeUsed` parameter is from the `routeSelected` parameter of the previous `cstaRouteSelect()` request received by the TSAPI Service for this call.
- The number provided in the `callingDevice` parameter is from the `callingDevice` parameter of the previous `CSTARouteRequestEvent` sent by the TSAPI Service for this call.

Syntax

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

/* CSTARouteUsedEvent - Route Select Service Response */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAEVENTREPORT */
    EventType_t      eventType;          /* CSTA_ROUTE_USED */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            union
            {
                CSTARouteUsedEvent_t routeUsed;
            } u;
        } cstaEventReport;
    } event;
} CSTAEvent_t;

typedef struct CSTARouteUsedEvent_t {
    RouteRegisterReqID_t routeRegisterReqID;
    RoutingCrossRefID_t routingCrossRefID;
    DeviceID_t routeUsed;
    DeviceID_t callingDevice;
    unsigned char domain;
} CSTARouteUsedEvent_t;
```


Chapter 13: System Status Service Group

The System Status Services Group provides 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](#) on page 962
- [System Status Start Service](#) on page 969
- [System Status Stop Service](#) on page 978
- [Change System Status Filter Service](#) on page 980
- [System Status Event](#) on page 989

Overview

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 and later), `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:

None for this service.

Ack Parameters:

`systemStatus`

[mandatory – partially supported] Provides the application with a cause code defining the overall system status as follows:

`SS_NORMAL` – This status indication indicates that the CTI link to the switch is available. The system status is normal, and TSAPI requests and responses are enabled.

`SS_DISABLED` – This system status indicates that there is no available CTI link to the switch. The `SS_DISABLED` status implies that there are no active Monitor requests or Route Register sessions. TSAPI requests that are dependent on Communication Manager are disabled, and will fail.

Ack Private Parameters:

<code>count</code>	Identifies the number of CTI links described in the private ack parameter <code>plinkStatus</code> (private data versions 5 and later) or <code>linkStatus</code> (private data versions 2-4). For AE Services, this number is always one.
<code>plinkStatus</code>	Specifies the status of each CTI link to the switch. For AE Services, the TSAPI Service supports a single CTI link to the switch, although this CTI link may be administered to use multiple CLAN cards. The routing of TSAPI service requests and responses over individual CLAN cards is hidden from the application. (A TSAPI application programmer does not need to consider the individual CLAN connections to the switch when sending/receiving TSAPI service requests/responses.) The <code>plinkStatus</code> private data parameter may be used to check the availability of the administered CTI link. The status of the link will be provided in the <code>linkState</code> field: • <code>LS_LINK_UP</code> – The link is able to support telephony services to the switch. • <code>LS_LINK_DOWN</code> – The link is unable to support telephony services to the switch. • <code>LS_LINK_UNAVAIL</code> – 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.
<code>linkStatus</code>	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 Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code> . The <code>error</code> parameter in this event may contain one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997.

Detailed Information:

- Multiple CLAN Connections – If multiple CLAN connections are connected and administered for a specific switch, the `systemStatus` parameter will indicate the aggregate link status. If at least one CLAN connection is available to support TSAPI requests and responses, the `systemStatus` will be set to `SS_NORMAL`. If there are no CLAN or Processor Ethernet connections to a switch able to support TSAPI requests and responses, the `systemStatus` will be set to `SS_DISABLED`.

Syntax

```

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

/* cstaSysStatReq() - Service Request */

RetCode_t    cstaSysStatReq(
    ACSHandle_t    acsHandle,
    InvokeID_t     invokeID,
    PrivateData_t  *privateData);

/* CSTASysStatReqConfEvent - Service Response */

typedef struct
{
    ACSHandle_t    acsHandle;
    EventClass_t   eventClass;          /* CSTACONFIRMATION */
    EventType_t    eventType;          /* CSTA_SYS_STAT_REQ_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            union
            {
                CSTASysStatReqConfEvent_t    sysStatReq;
            } u;
        } cstaConfirmation;
    } 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 and Later Syntax

```

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

/* ATTLinkStatusEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATT_LINK_STATUS */
    union
    {
        ATTLinkStatusEvent_t    linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTLinkStatusEvent_t {
    unsigned int          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 - Service Response Private Data */

typedef struct
{
    ATTEventType_t    eventType;          /* ATTV4_LINK_STATUS */
    union
    {
        ATTV4LinkStatusEvent_t    v4linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTV4LinkStatusEvent_t {
    unsigned 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_t    eventType;          /* ATTV3_LINK_STATUS */
    union
    {
        ATTV3LinkStatusEvent_t    v3linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTV3LinkStatusEvent_t {
    unsigned 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 and later), `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 TSAPI CTI link 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 active `cstaSysStatStart()` request is allowed for an `acsOpenStream()` request. If a `cstaSysStatStart()` request is active, the second request will be rejected.

Service Parameters:

<code>statusFilter</code>	[mandatory – partially supported] A filter used to specify the system status events that are not of interest to the application. If a bit in <code>statusFilter</code> is set to <code>TRUE (1)</code> , the corresponding event will not be sent to the application. The only System Status events that are supported are <code>SS_ENABLED</code> , <code>SS_NORMAL</code> and <code>SS_DISABLED</code> . 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 CTI links. This capability is a holdover from the Avaya Computer-Telephony product, and is not useful for AE Services configurations. The Avaya Computer-Telephony product allowed multiple CTI links to be configured between the Telephony Server and Avaya Communication Manager. For AE Services, only a single TSAPI CTI link may be configured for any given switch; the use of multiple CLAN cards to support that switch connection is hidden from the TSAPI Service.

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 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:

<code>count</code>	Identifies the number of CTI links described in the private ack parameter <code>plinkStatus</code> (private data versions 5 or later) or <code>linkStatus</code> (private data versions 2-4). This parameter is only provided when the <code>linkStatusReq</code> private parameter was set to <code>TRUE</code> . For AE Services, this number is always one.
<code>plinkStatus</code>	Specifies the status of each CTI link to the switch. For AE Services, the TSAPI Service supports a single CTI link to the switch, although this CTI link may be administered to use multiple CLAN cards. This parameter is only provided when the <code>linkStatusReq</code> private parameter was set to <code>TRUE</code>. The <code>plinkStatus</code> private data parameter will indicate the availability of the administered CTI link to which the application is connected. The status of the link will be provided in the <code>linkState</code> field: • <code>LS_LINK_UP</code> – The link is able to support telephony services to the switch. • <code>LS_LINK_DOWN</code> – The link is unable to support telephony services to the switch. • <code>LS_LINK_UNAVAIL</code> -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.
<code>linkStatus</code>	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 Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The <code>error</code> parameter in this event may contain the following error value, or one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997. • <code>GENERIC_OPERATION_REJECTION (71)</code> – Only one active <code>cstaSysStatStart()</code> request is allowed for an <code>acsOpenStream()</code> request. If an active request exists, the second request will be rejected.

Detailed Information:

- The `linkStatReq` private parameter is no longer useful for AE Services; it is a holdover from the Avaya Computer Telephony product.
- Only one active `cstaSysStatStart()` request is allowed for an `acsOpenStream()` request. If an active request exists, the second request will be rejected. An application can cancel a request for System Status event reporting via `cstaSysStatStop()`, and then issue a subsequent `cstaSysStatStart()` request.
- 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 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 the CTI link is down (`LS_LINK_DOWN`) or unavailable (`LS_LINK_UNAVAIL`). 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 CTI link to Communication Manager has been established. 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 CTI link to Communication Manager has been established. For AE Services, this event is redundant with the `CSTASysStatEvent` Event Report just received with the `systemStatus` set to `SS_ENABLED`.

Syntax

```

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

/* cstaSysStatStart() - Service Request */

RetCode_t    cstaSysStatStart(
    ACSHandle_t      acsHandle,
    InvokeID_t       invokeID,
    SystemStatusFilter_t  statusFilter,
    PrivateData_t    *privateData);

typedef unsigned  SystemStatusFilter_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_t     eventClass;          /* CSTACONFIRMATION */
    EventType_t      eventType;          /* CSTA_SYS_STAT_START_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTASysStatStartConfEvent_t  sysStatStart;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASysStatStartConfEvent_t {
    SystemStatusFilter_t    statusFilter;

```

```
} CSTASysStatStartConfEvent_t;
```

Private Data Version 5 and Later Syntax

```

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

/* attSysStat() - Service Request Private Data Formatting Function */

RetCode_t attSysStat(
    ATTPrivateData_t      *privateData,
    Boolean                linkStatusReq);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTLinkStatusEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t      eventType; /* ATT_LINK_STATUS */
    union
    {
        ATTLinkStatusEvent_t linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTLinkStatusEvent_t {
    unsigned int          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 Formatting Function */

RetCode_t attSysStat(
    ATTPrivateData_t      *privateData,
    Boolean                linkStatusReq);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTV4LinkStatusEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t      eventType;          /* ATTV4_LINK_STATUS */
    union
    {
        ATTV4LinkStatusEvent_t      v4linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTV4LinkStatusEvent_t {
    unsigned 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 Formatting Function */

RetCode_t attSysStat(
    ATTPrivateData_t      *privateData,
    Boolean                linkStatusReq);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTV3LinkStatusEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t      eventType;          /* ATTV3_LINK_STATUS */
    union
    {
        ATTV3LinkStatusEvent_t      v3linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTV3LinkStatusEvent_t {
    unsigned 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:

None for this service.

Ack Parameters:

None for this service.

Nak Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code> . The <code>error</code> parameter in this event may contain one of the error values described in Table 21: Common switch-related CSTA Service errors -- universalFailure on page 997.

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_t   eventClass;           /* CSTACONFIRMATION */
    EventType_t    eventType;           /* CSTA_SYS_STAT_STOP_CONF
*/
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTASysStatStopConfEvent_t    sysStatStop;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTASysStatStopConfEvent_t {
    Nulltype    null;
} CSTASysStatStopConfEvent_t;

typedef char Nulltype;

```

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 and later), `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()` request will be rejected.

Service Parameters:

<code>statusFilter</code>	[mandatory – partially supported] A filter used to specify the System Status Events that are not of interest to the application. If a bit in <code>statusFilter</code> is set to <code>TRUE (1)</code> , the corresponding event will not be sent to the application. The only System Status Events that are supported are <code>SS_ENABLED</code> , <code>SS_NORMAL</code> and <code>SS_DISABLED</code> . 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 CTI links. This capability is a holdover from the Avaya Computer-Telephony product, and is not useful for AE Services configurations.

The Avaya Computer-Telephony product allowed multiple CTI links to be configured between the Telephony Server and Avaya Communication Manager. For AE Services, only a single TSAPI CTI link may be configured for any given switch; the use of multiple CLAN cards to support that switch connection is hidden from the TSAPI Service.

- 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 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 CTI links. (If System Status Event Reports were sent for changes in individual CTI links before a `cstaChangeSysStatFilter()` 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 `CSTChangeSysStatFilterConfEvent` 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:

<code>count</code>	Identifies the number of CTI links described in the private ack parameter <code>plinkStatus</code> (private data versions 5 or later) or <code>linkStatus</code> (private data versions 2-4). This parameter is only provided when the <code>linkStatusReq</code> private parameter was set to <code>TRUE</code> . For AE Services, this number is always one.
<code>plinkStatus</code>	Specifies the status of each CTI link to the switch. For AE Services, the TSAPI Service supports a single CTI link to the switch, although this CTI link may be administered to use multiple CLAN cards. This parameter is only provided when the <code>linkStatusReq</code> private parameter was set to <code>TRUE</code>. The <code>plinkStatus</code> private data parameter will indicate the availability of the administered CTI link to which the application is connected. The status of the link will be provided in the <code>linkStatus</code> field: • <code>LS_LINK_UP</code> – The link is able to support traffic. • <code>LS_LINK_DOWN</code> – The link is unable to support traffic. • <code>LS_LINK_UNAVAIL</code> – 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.
<code>linkStatus</code>	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 Parameters:

<code>universalFailure</code>	If the request is not successful, the application will receive a <code>CSTAUniversalFailureConfEvent</code>. The <code>error</code> parameter in this event may contain the following error value, or one of the error values described in Table 21: Common switch-related CSTA Service errors - universalFailure on page 997. • <code>GENERIC_OPERATION_REJECTION (71)</code> – If the application has not registered to receive System Status Event reports, the <code>cstaChangeSysStatFilter()</code> request will be rejected.

Detailed Information:

- The `linkStatReq` private parameter is no longer useful for AE Services; it is a holdover from the Avaya Computer Telephony product.
- 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.
- 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  SystemStatusFilter_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 */

/* 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
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            InvokeID_t  invokeID;
            union
            {
                CSTAChangeSysStatFilterConfEvent_t
changeSysStatFilter;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAChangeSysStatFilterConfEvent_t {

```

```

        SystemStatusFilter_t      statusFilterSelected;
        SystemStatusFilter_t      statusFilterActive;
    } CSTAChangeSysStatFilterConfEvent_t;

```

Private Data Version 5 and Later Syntax

```

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

/* attSysStat() - Service Request Private Data Formatting Function */

RetCode_t attSysStat(
    ATTPrivateData_t      *privateData,
    Boolean                linkStatusReq);

typedef struct ATTPrivateData_t {
    char                  vendor[32];
    unsigned short        length;
    char                  data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTLinkStatusEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t        eventType; /* ATT_LINK_STATUS */
    union
    {
        ATTLinkStatusEvent_t linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTLinkStatusEvent_t {
    unsigned int           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 Formatting Function */

RetCode_t attSysStat(
    ATTPrivateData_t      *privateData,
    Boolean                linkStatusReq);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTV4LinkStatusEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t      eventType;          /* ATTV4_LINK_STATUS */
    union
    {
        ATTV4LinkStatusEvent_t      v4linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTV4LinkStatusEvent_t {
    unsigned 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 Formatting Function */

RetCode_t attSysStat(
    ATTPrivateData_t      *privateData,
    Boolean                linkStatusReq);

typedef struct ATTPrivateData_t {
    char                vendor[32];
    unsigned short      length;
    char                data[ATT_MAX_PRIVATE_DATA];
} ATTPrivateData_t;

/* ATTV3LinkStatusEvent - Service Response Private Data */

typedef struct
{
    ATTEventType_t      eventType;          /* ATTV3_LINK_STATUS */
    union
    {
        ATTV3LinkStatusEvent_t      v3linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTV3LinkStatusEvent_t {
    unsigned 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 and later), `linkStatus` (private data versions 2, 3, and 4)

Functional Description:

This unsolicited event is sent by the TSAPI Service 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 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 CTI link to Communication Manager has been established. 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 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 CTI link changes state to up (`LS_LINK_UP`). The `systemStatus` normal (`SS_NORMAL`) indicates that the CTI link to the switch is available.

Private Parameters:

<code>count</code>	Identifies the number of CTI links described in the private ack parameter <code>plinkStatus</code> (private data version 5 and later) or <code>linkStatus</code> (private data versions 2-4). This parameter is only provided when the <code>linkStatusReq</code> private parameter was set to TRUE. For AE Services, this number is always one.
<code>plinkStatus</code>	Specifies the status of each CTI link to the switch. For AE Services, the TSAPI Service supports a single CTI link to the switch, although this CTI link may be administered to use multiple CLAN cards. This parameter is only provided when the <code>linkStatusReq</code> private parameter was set to TRUE. The <code>plinkStatus</code> private data parameter will indicate the availability of the administered CTI link to which the application is connected. The status of the link will be provided in the <code>linkStatus</code> field: • <code>LS_LINK_UP</code> – The link is able to support telephony services to the switch. • <code>LS_LINK_DOWN</code> – The link is unable to support telephony services to the switch. • <code>LS_LINK_UNAVAIL</code> – 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.
<code>linkStatus</code>	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:

- When the CTI link is established, a System Status Event Report will be sent to the application with the `systemStatus` set to `SS_ENABLED`, followed by a System Status Event Report with the `systemStatus` set to `SS_NORMAL`. When the CTI link fails, a System Status Event Report will be sent to the application with the `systemStatus` set to `SS_DISABLED`.

Syntax

```

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

/* CSTASysStatEvent - System Status Event */

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTAEVENTREPORT */
    EventType_t      eventType;          /* CSTA_SYS_STAT */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t  eventHeader;
    union
    {
        struct
        {
            union
            {
                CSTASysStatEvent_t      sysStat;
            } u;
        } cstaEventReport;
    } event;
} CSTAEvent_t;

typedef struct CSTASysStatEvent_t {
    SystemStatus_t      systemStatus;
} CSTASysStatEvent_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 and Later Syntax

```

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

/* ATTLinkStatusEvent - System Status Event Private Data */

typedef struct
{
    ATTEventType_t    eventType; /* ATT_LINK_STATUS */
    union
    {
        ATTLinkStatusEvent_t    linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTLinkStatusEvent_t {
    unsigned int          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_t    eventType;          /* ATTV4_LINK_STATUS */
    union
    {
        ATTV4LinkStatusEvent_t    v4linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTV4LinkStatusEvent_t {
    unsigned 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_t    eventType;          /* ATTV3_LINK_STATUS */
    union
    {
        ATTV3LinkStatusEvent_t    v3linkStatus;
    } u;
} ATTEvent_t;

typedef struct ATTV3LinkStatusEvent_t {
    unsigned 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 provides the following error summaries:

- [Common switch-related CSTA Service errors](#) on page 996
- [TSAPI Client library error codes](#) on page 1003
- [ACSUniversalFailureConfEvent error values](#) on page 1007
- [ACS Related Errors](#) on page 1025

Common switch-related CSTA Service errors

[Table 21](#) lists the most commonly used CSTA errors returned by CSTA Services in the `CSTAUniversalFailureConfEvent` for a negative acknowledgment to 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 (rather than the switch driver). Those error codes are enumerated in [Syntax](#) on page 1001.

An application program should be able to handle any CSTA error defined by `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 21: Common switch-related CSTA Service errors -- universalFailure

Error	Description
GENERIC_UNSPECIFIED (0)	An error has occurred. The TSAPI Service could not provide one of the more specific error values described below.
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 16.
REQUEST_INCOMPATIBLE_WITH_OBJECT (2)	The service request does not correspond to a CTI application association. Report this error -- Customer Support on page 16.
VALUE_OUT_OF_RANGE (3)	Communication Manager detects that a required parameter is missing from 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 <code>deviceID</code> of a <code>connectionID</code> 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.
GENERIC_SUBSCRIBED_RESOURCE_AVAILABILITY (41)	The TSAPI Service could not acquire the license(s) needed to satisfy the request.
OUTSTANDING_REQUEST_LIMIT_EXCEEDED (44)	The given request cannot be processed due to a system resource limit on the device.

Table 21: Common switch-related CSTA Service errors -- universalFailure

Error	Description
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 16.
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 16.
DUPLICATE_INVOCATION_REJECTION (72)	The TSAPI Service detects that the <code>invokeID</code> in the service request is being used by another outstanding service request. This service request is rejected. The outstanding service request with the same <code>invokeID</code> 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 <code>invokeID</code> 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.

Table 21: Common switch-related CSTA Service errors -- universalFailure

Error	Description
ACS_HANDLE_TERMINATION_REJECTION (76)	The TSAPI Service detects that an <code>acsOpenStream</code> 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 <code>acsCloseStream</code> request and waits for the confirmation event. In this case, the <code>acsCloseStream</code> 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 <code>acsCloseStream</code> 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.
REQUEST_TIMEOUT_REJECTION (78)	The TSAPI Service did not receive the response of a service request sent to Communication Manager more than 30 seconds ago. The timer created for 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 out of service or congested. Congestion may occur when TSAPI applications exceed the capacity of the TSAPI Service.

Table 21: Common switch-related CSTA Service errors -- universalFailure

Error	Description
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 16. 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 <code>MAX_REQS_QUEUED_PER_DEVICE</code> parameter has no effect on this class of applications. Situations of sending a sequence of requests without waiting for their completion are rare. However, if this is the case, the <code>MAX_REQS_QUEUED_PER_DEVICE</code> parameter should be set to a proper value. The default value for <code>MAX_REQS_QUEUED_PER_DEVICE</code> is 4.

Syntax

The following structure shows only the relevant portions of the unions for this message:

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

typedef struct
{
    ACSHandle_t      acsHandle;
    EventClass_t     eventClass;          /* CSTACONFIRMATION */
    EventType_t      eventType;          /*
CSTA_UNIVERSAL_FAILURE_CONF */
} ACSEventHeader_t;

typedef struct
{
    ACSEventHeader_t eventHeader;
    union
    {
        struct
        {
            InvokeID_t invokeID;
            union
            {
                CSTAUniversalFailureConfEvent_t
                universalFailure;
            } u;
        } cstaConfirmation;
    } event;
} CSTAEvent_t;

typedef struct CSTAUniversalFailureConfEvent_t {
    CSTAUniversalFailure_t error;
} CSTAUniversalFailureConfEvent_t;

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,
    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 errors listed above may be provided by either the switch
 * or by the TSAPI Service.
 * The following errors are only provided by the TSAPI Service
 * and 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,
UNRECOGNIZED_APDU_REJECTION = 80,
MISTYPED_APDU_REJECTION = 81,
BADLY_STRUCTURED_APDU_REJECTION = 82,
INITIATOR_RELEASING_REJECTION = 83,
UNRECOGNIZED_LINKEDID_REJECTION = 84,
LINKED_RESPONSE_UNEXPECTED_REJECTION = 85,
UNEXPECTED_CHILD_OPERATION_REJECTION = 86,
MISTYPED_RESULT_REJECTION = 87,
UNRECOGNIZED_ERROR_REJECTION = 88,
UNEXPECTED_ERROR_REJECTION = 89,
MISTYPED_PARAMETER_REJECTION = 90,
NON_STANDARD = 100
} CSTAUniversalFailure_t;

```


TSAPI Client library error codes

[Table 22](#) describes TSAPI Client library 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 22: TSAPI Client Library 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.	Does the TSAPI library configuration file (TSLIB.INI or tslibrc) contain the correct server name or IP address for the AE Services server? When using host names instead of IP addresses, can the host name be resolved to an IP address? Is the TSAPI Service up (online)? Are physical network connections (wiring) intact?
-6	This return value indicates that there are insufficient resources to open a connection.	Contact the application developer. The application may be trying to open too many connections or may be 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)	Is host name resolution properly configured on the AE Services server, such that the server is able to resolve the IP address of the client machine to a host name? Call Customer Support and report this error. See Customer Support on page 16.
-9	The connection has encountered an unspecified error.	This error is often the result of a software version mismatch. Has some software been replaced or upgraded recently? Call Customer Support and report this error. See Customer Support on page 16.

Table 22: TSAPI Client Library Error Codes

Error	Description	Corrective Action
-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 whether the TSAPI Service is running. From the AE Services Management Console (OAM), select Status > Status and Control > TSAPI Service Summary . Also, check that physical network connections are intact.
-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.	Consult the application developer.
-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.	Consult the application developer.
-14	This return value indicates that a secure connection could not be opened because there was a problem initializing the OpenSSL library.	See Customer Support on page 16.

Table 22: TSAPI Client Library Error Codes

Error	Description	Corrective Action
-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 with valid key usage or extended key usage values or that the server certificate is either not provided or not signed by a trusted Certificate Authority.	For AE Services 6.3.x clients or older, this error may occur when trying to establish a secure TLS connection to the AE Services 7.0.1 server in which the TLS 1.2 protocol is enabled by default. The TLS 1.2 protocol is supported in the AE Services 7.0 SDK. If upgrading to the AE Services 7.0 SDK is not currently feasible, TLS 1.0 can be enabled via the AE Services Management Console interface (under Networking->TCP/TLS Settings). For AE Services 7.0 Windows clients, this error may occur when trying to establish a secure TLS connection to the AE Services 7.0.1 server when the OpenSSL library does not support TLS 1.2. The AE Services 7.0 TSAPI client that support TLS 1.2 is tsapi-client-win32-7.0.0-138. Verify this version (or newer) is being used. See AE Services 8.1.3 clients and backward compatibility on page 12. Beginning in AE Services 7.0 server, the default certificate is replaced with a one-year self-signed server certificate. Check with your network administrator to determine whether the AE Services server certificate is using the default certificate for TSAPI client connections. If the default self-signed certificate is being used, you will need to export the server certificate from AE Services and use it for secure connections. To do so, edit the TSAPI library configuration file (TSLIB.INI or tslibrc) and change the "Trusted CA File" setting in the "[Config]" section to the full path name of the appropriate trusted CA certificate.

Table 22: TSAPI Client Library Error Codes

Error	Description	Corrective Action
-16	This return value indicates that a stream could not be opened because the Fully Qualified Domain Name (FQDN) in the server certificate does not match the expected FQDN.	Check with your network administrator to determine whether the AE Services server certificate contains the correct FQDN for the AE Services server. If not, edit the TSAPI library configuration file (TSLIB.INI or tslibrc) and change the "Verify Server FQDN" setting in the "[Config]" section from "1" to "0".

ACSUniversalFailureConfEvent error values

Error values in this category indicate that the TSAPI Service detected an ACS-related error. [Table 23](#) describes ACS Universal Failure event error codes.

➤ NOTE:

An `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` (the unsolicited version of this confirmation event) is sent to the application, see [ACSUniversalFailureEvent](#) on page 111.

Table 23: 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 16.
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 23: ACS Universal Failure Events

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 CRITICAL 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.

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
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.
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 16.

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
12	TSERVER_ENCODE_FAILED	TSAPI Service	The TSAPI Service was unable to encode a message from the G3PD to a client workstation.	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 16.
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 16.
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 16.
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 16).

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
16	TSERVER_NO_VERSION	TSAPI Service	The TSAPI Service received an ACSOpenStreamConf-Event from the G3PD which does not have one of the version fields set correctly. The confirmation event 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 16).
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.
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 16.
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 16.
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 16.

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
23	TSERVER_BAD_SERVERID	SDB	The TSAPI Service rejected an <code>acsOpenStream</code> 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 the TSAPI 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 <code>acsOpenStream</code> request was invalid.	A software problem has occurred with the client library. Call Customer Support and report this error. See Customer Support on page 16.
25	TSERVER_BAD_PASSWORD_OR_LOGIN	SDB	The password, login, or both from an <code>acsOpenStream</code> 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 <code>acsOpenStream</code> 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 (Security > Security Database > Devices) Note: Make sure the assigned Tlink group for this device includes the correct Tlink.

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
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_RESTRICTED_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.
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 16.
35	TSERVER_DEVICE_NOT_SUPPORTED	SDB	The Tlink group administered for this device does not contain the CTI link to which the user opened a 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 (Security > Security Database > Devices) 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.

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
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 16.
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.
40	TSERVER_NOT_ON_OAM_LIST	TSAPI Service	Obsolete message.	Obsolete message.
41	TSERVER_PBXID_NOT_IN_SDB	TSAPI Service	Obsolete message.	Obsolete message.
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	Obsolete message.	Obsolete message.
45	TSERVER_OLD_VERSION_RECORD	TSAPI Service	Obsolete message.	Obsolete message.
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 a connection, so the connection was dropped.	An error code should have been returned in response to the <code>acsOpenStream()</code> request in the <code>ACSEventFailureConfEvent</code> . Follow the procedures defined for that error code.

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
48	TSERVER_OAM_IN_USE	TSAPI Service	Obsolete message.	Obsolete message.
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 Device Group in this user's record which corresponds to the action being attempted ("Call Origination/Termination and Device Status" or "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 (Security > Security Database > Worktops) • Edit the user's "Access Rights" (Security > Security Database > CTI Users > Edit CTI User).

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
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 in the Security Database so that the user has permission to control the device through either the worktop object (Worktop administration) or through the "Call Origination/Termination and Device Status" Device Group (CTI User administration).
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 Device Group in this user's record which corresponds to the action being attempted ("Call Origination/Termination and Device Status" or "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 in the Security Database so that the user has permission to control the device through either the user's worktop object (Worktop administration) or through the appropriate Device Group (CTI User administration).

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
52	TSERVER_DEVICE_NOT_ON_- ROUTE_LIST	SDB	The telephony server has rejected a user's routing request for a device because the user is assigned a "Routing Control" Device Group, but the device is not a member of that group.	Change the user's administration in the Security Database so that the user has permission to control the device through the "Routing Control" Device Group (Device Group administration).
53	TSERVER_DEVICE_NOT_ON_- MONITOR_DEVICE_LIST	SDB	The telephony server rejected a user's monitor device request because the user is assigned a "Device Monitoring" Device Group, but the device is not a member of that group.	Change the user's administration in the Security Database so that the user has permission to control the device through either the worktop record (Worktop administration) or through the "Device Monitoring" Device Group (Device Group 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 in the user's "Calls On A Device Monitoring" Device Group and the "Calls On A Device Monitoring" Device Group is not empty.	Change the user's administration in the Security Database so that the user has permission to control the device through the "Calls On A Device Monitoring" Device Group (Device Group administration).
55	TSERVER_NO_CALL_CALL_- MONITOR_PERMISSION	SDB	The telephony server rejected a user's request to monitor a call because the Call Monitoring is not enabled for this user.	Change the user's administration in the Security Database to enable Call Monitoring (CTI User administration).
56	TSERVER_HOME_DEVICE_LIST_- EMPTY	SDB	Obsolete message.	Obsolete message.
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 Control" Device Group in this user's record is empty.	Change the user's administration in the Security Database so that the user has permission to control the device through the "Routing Control" Device Group (Device Group administration).
60	TSERVER_MONITOR_DEVICE_- LIST_EMPTY	SDB	Obsolete message.	Obsolete message.

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
61	TSERVER_MONITOR_CALL_- DEVICE_LIST_EMPTY	SDB	The telephony server rejected a user's request to control a device because the "Calls On A Device Monitoring" Device Group in this user's record is empty.	Change the user's administration in the Security Database so that the user has permission to control the device through the "Calls On A Device Monitoring Device Group" (Device Group 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 user record are empty (in the set of lists searched for this type of message).	Change the user's administration in the Security Database so that the user has permission to control the device through either the user's worktop record (Worktop administration) or through the appropriate Device Group (CTI User administration).
64	TSERVER_BAD_GET_DEVICE_- LEVEL	SDB	A <code>cstaGetDeviceList</code> query was made with a bad <code>CSTALevel_t</code> value. Valid <code>CSTALevels</code> are: • <code>CSTA_HOME_WORK_-TOP</code> • <code>CSTA_AWAY_WORK_-TOP</code> • <code>CSTA_DEVICE_-DEVICE_MONITOR</code> • <code>CSTA_CALL_DEVICE_MONITOR</code> • <code>CSTA_CALL_CONTROL</code> • <code>CSTA_ROUTING</code>	The application has called <code>cstaGetDeviceList</code> 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 16.

Table 23: ACS Universal Failure Events

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 <code>acsOpenStream</code> request or violated the protocol by not returning an <code>ACSOpenStreamConfEvent</code> .	1. The TSAPI Service will terminate this stream when this error occurs. Verify that the application waits for an <code>ACSOpenStreamConfEvent</code> 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 16.
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 an 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 16.
71	TSERVER_ADVERTISE_FAILED	TSAPI Service	Obsolete message.	Obsolete message.
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 16.

Table 23: ACS Universal Failure Events

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 has reached the highwater mark. This occurs when the TSAPI Service is not processing messages fast enough.	1. Increase the TSDI space. In AE Services > TSAPI > TSAPI Links > Edit Link > Advanced Settings. 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 AE Services > TSAPI > TSAPI Links > Edit Link > Advanced Settings. 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 <code>invokeID</code> 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 23: ACS Universal Failure Events

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 (<code>apiVer</code> field in <code>acsOpenStream()</code>) set to a value that the TSAPI Service does not support.	From the client workstation, use the TSAPI Spy to determine what protocol version(s) the application is requesting in the <code>acsOpenStream()</code> request. Currently, the supported protocol versions are 1 and 2.
82	TSERVER_TSERVER_BAD_- DRIVER_PROTOCOL	TSAPI Service	A client application attempted to open a stream with a protocol version (<code>apiVer</code> field in <code>acsOpenStream()</code>) set to a value that the PBX Driver the stream was destined for does not support.	Use Status > Status and Control > TSAPI Service Summary . From the 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_BAD_TRANSPORT_TYPE	TSAPI Service	Obsolete message.	Obsolete message.
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 Status > Status and Control > TSAPI Service Summary . From the 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.
86	TSERVER_LICENSE_MISMATCH	TSAPI Service	Obsolete message.	Obsolete message.
87	TSERVER_BAD_ATTRIBUTE_LIST	TSAPI Service	Obsolete message.	Obsolete message.

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
88	TSERVER_BAD_TLIST_TYPE	TSAPI Service	Obsolete message.	Obsolete message.
89	TSERVER_BAD_PROTOCOL_FORMAT	TSAPI Service	A client application attempted to open a stream with a protocol version (<code>apiVer</code> field in <code>acsOpenStream()</code>) 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 16.
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 16.
96	TSERVER_SPX_DISABLED	TSAPI Service	Obsolete message.	Obsolete message.
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.

Table 23: ACS Universal Failure Events

Error	Message	Type	Description	Corrective Action
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 shared object files are located in the proper directory. 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 23: ACS Universal Failure Events

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.	This is an application error. The application has invoked <code>acsSetHeartbeatInterval()</code> with an invalid value. The valid range of values is 5 to 60 (seconds).

ACS Related Errors

Table 24: ACS Related Errors

Error	Message	Type	Description
1000	DRIVER_DUPLICATE_ACSHANDLE	TSAPI Service	The ACS Handle given for an ACS Stream request is already in use for a session. The already open session with the ACS Handle 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 <code>acsOpenStream()</code> request, or the ACS Handle given for an <code>acsOpenStream()</code> 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 16.
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 16.
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 ACS Handle. 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 <code>BLOCKED</code> 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 <code>acsOpenStream()</code> 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 has requested system status event reports via the <code>cstaSysStatStart()</code> service.

Appendix B: Summary of Private data support

This appendix provides historical information about private data versions in the current and previous releases.

Private Data Version 17 features

The TSAPI Service supports private data version 17 beginning with AE Services Release 8.1.3. Private data version 17 supports Avaya proprietary features.

Private Data Version 16 features

The TSAPI Service supports private data version 16 beginning with AE Services Release 8.1.2. Private data version 16 supports Avaya proprietary features.

Private Data Version 15 features

The TSAPI Service supports private data version 15 beginning with AE Services Release 8.1. Private data version 15 provides the following new features:

Channel Type in Originated event

Beginning with private data version 15, the private data accompanying the CSTA Originated Event includes the Channel Type, if available.

VideoCall Option

Beginning with private data version 15, the private data accompanying the CSTA Make Call, CSTA Answer Call, CSTA Consultation Call and ATT Single Step Transfer Call includes the VideoCall Type, if available. VideoCall option only support for Avaya Aura Media Server(AMS).

Private Data Version 14 features

The TSAPI Service supports private data version 14 beginning with AE Services Release 7.1.3.1. Private data version 14 supports Avaya proprietary features.

Private Data Version 13 features

The TSAPI Service supports private data version 13 beginning with AE Services Release 7.1.1. Private data version 13 provides the following new feature:

Channel Type in Delivered Event

Beginning with private data version 13, the private data accompanying the CSTA [Diverted Event](#) includes the Channel Type, if available.

Channel Type in Established Event

Beginning with private data version 13, the private data accompanying the CSTA [Diverted Event](#) includes the Channel Type, if available.

Private Data Version 12 features

The TSAPI Service supports private data version 12 beginning with AE Services Release 6.3.3. Private data version 12 provides the following new feature:

Calling Device in Diverted Event

Beginning with private data version 12, the private data accompanying the CSTA [Diverted Event](#) includes the Calling Device, if available.

Private Data Version 11 features

The TSAPI Service supports private data version 11 beginning with AE Services Release 6.3.1. Private data version 11 provides the following new features:

Endpoint Information Query

Beginning with private data version 11, the Query Endpoint Registration Info service (`attQueryEndpointRegistrationInfo()`) allows an application to query for the service state of a station extension and for a list of H.323 and/or SIP endpoints registered to that station extension.

The Query Endpoint Registration Info service is available beginning with Avaya Communication Manager Release 6.3.2 and AE Services Release 6.3.1.

This service is only available if the TSAPI Link is administered with ASAI Link Version 6 or later and if the application has negotiated private data version 11 or later.

Applications should use the `cstaGetAPICaps()` service and check the value of the `queryEndpointRegistrationInfo` parameter in the private data accompanying the CSTA Get API Caps Confirmation event to determine whether this service is available.

Applications should invoke `cstaMonitorDevice()` for the station extension before invoking this query because Avaya Communication Manager cannot provide the service state of a SIP station that is not monitored.

Endpoint Registered Event

Beginning with private data version 11, an Endpoint Registered event is sent when an H.323 or SIP endpoint registers to a monitored station extension.

This event is available beginning with Communication Manager Release 6.3.2 and AE Services Release 6.3.1.

This event is only available if the TSAPI Link is administered with ASAI Link Version 6 or later and if the application has negotiated private data version 11 or later.

Applications should use the `cstaGetAPICaps()` service and check the value of the `endpointRegisteredEvent` parameter in the private data accompanying the CSTA Get API Caps Confirmation event to determine whether this event will be provided.

Endpoint Unregistered Event

Beginning with private data version 11, an Endpoint Unregistered event is sent when an H.323 or SIP endpoint unregisters from a monitored station extension.

This event is available beginning with Communication Manager Release 6.3.2 and AE Services Release 6.3.1.

This event is only available if the TSAPI Link is administered with ASAI Link Version 6 or later and if the application has negotiated private data version 11 or later.

Applications should use the `cstaGetAPICaps()` service and check the value of the `endpointUnregisteredEvent` parameter in the private data accompanying the CSTA Get API Caps Confirmation event to determine whether this event will be provided.

Failed Event Device Identifiers

Beginning with private data version 11, device identifiers reported in the CSTA Failed event and ATT Failed event are set differently than for earlier private data versions.

Previously, in most scenarios the TSAPI Service would set the following event parameters using the `deviceId` of the `calledDevice`:

- the `failedConnection deviceId`
- the `failingDevice deviceId`
- the device history `olddeviceId`
- the device history `oldconnectionID deviceId`

However, for some call scenarios, this is incorrect.

Because the TSAPI Service cannot always provide the correct device ID in these event parameters, beginning with private data 11 the TSAPI Service sets these parameters differently for some call scenarios. In those scenarios:

- the `failedConnection deviceId` is set to the empty string (“”)
- the `failingDevice deviceId` is set to the empty string (“”) and its device ID status is reported as `ID_NOT_KNOWN`
- the device history `olddeviceId` is set to “Not Known”
- the device history `oldconnectionID deviceId` is set to “Not Known”.

Private Data Version 11 features, services, and events

The following table maps the Private Data Version 11 features to the services and events that they affect.

Table 25: Private Data Version 11 features, services, and events

Private Data Version 11 feature	Updated services and events
Endpoint Information Query	• Query Endpoint Registration Info Service (Private Data Version 11 and later) on page 514
Endpoint Registered Event	• Endpoint Registered Event (Private Data Version 11 and later) on page 737
Endpoint Unregistered Event	• Endpoint Unregistered Event (Private Data Version 11 and later) on page 744
Failed Event Called Device	• Failed Event on page 789

Private Data Version 10 features

The TSAPI Service supports private data version 10 beginning with AE Services Release 6.2. Private data version 10 provides the following new features:

- Agent Work Mode in Logged On Event
- Consult Options
- Enhanced Station Status Query

Agent Work Mode in Logged On Event

Beginning with private data version 10, private data accompanying the CSTA Logged On event provides the initial work mode of the agent. The possible work modes are:

- Auxiliary Work Mode (`WM_AUX_WORK`)
- After-Call Work Mode (`WM_AFTCAL_WK`)
- Auto-In Mode (`WM_AUTO_IN`)
- Manual-In Mode (`WM_MANUAL_IN`)

Consult Options

Beginning with private data version 10, the function `attV10ConsultationCall()` allows an application to indicate the intended purpose of a consultation call via the `consultOptions` parameter. The valid values for this parameter are:

- `CO_UNRESTRICTED`
- `CO_CONSULT_ONLY`
- `CO_TRANSFER_ONLY`
- `CO_CONFERENCE_ONLY`

When an application invokes the `cstaConsultationCall()` service with `consultOptions CO_TRANSFER_ONLY`, the private data consult mode in the corresponding Held, Service Initiated, and Originated events will be set to `ATT_CM_TRANSFER`.

Similarly, when an application invokes the `cstaConsultationCall()` service with `consultOptions CO_CONFERENCE_ONLY`, the private data consult mode in the corresponding Held, Service Initiated, and Originated events will be set to `ATT_CM_CONFERENCE`.

When an application invokes the `cstaConsultationCall()` service with `consultOptions CO_UNRESTRICTED` or `CO_CONSULT_ONLY`, or when an application invokes the `cstaConsultationCall()` service without using `attV10ConsultationCall()`, the private data consult mode in the corresponding Held, Service Initiated, and Originated events will be set to `ATT_CM_CONSULTATION`.

Enhanced Station Status Query

Beginning with private data version 10, the function `attV10QueryStationStatus()` allows an application to query for a station's service state (in-service or out-of-service) in addition to its talk state (busy or idle).

Private Data Version 10 features, services, and events

The following table maps the Private Data Version 10 features to the services and events that they affect.

Table 26: Private Data Version 10 features, services, and events

Private Data Version 10 feature	Updated services and events
Agent Work Mode in Logged On Event	• Logged On Event on page 810
Consult Options	• Consultation Call Service on page 283 • Held Event on page 802 • Originated Event on page 827 • Service Initiated Event on page 848
Enhanced Station Status Query	• Query Station Status Service on page 531

CSTA Get API Capabilities Private Data Versions 8 – 10 Syntax

The TSAPI Service provides version-dependent private services in the `CSTAGetAPICaps` Confirmation private data interface. For Private Data Version 8 the private data confirmation event was updated for the single step transfer call service: Within the private data confirmation event, the field `reserved1` was renamed `singleStepTransfer`. This field indicates whether the single step transfer call service is available.

CSTA Get API Capabilities Private Data Version 8-10 Syntax

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

typedef struct ATTV10GetAPICapsConfEvent_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   monitorCallsViaDevice;
    unsigned char   deviceHistoryCount;
    char            adminSoftwareVersion[256];
    char            softwareVersion[256];
    char            offerType[256];
    char            serverType[256];
} ATTV10GetAPICapsConfEvent_t;
```

Private Data Version 9 features

The TSAPI Service supports private data version 9 beginning with AE Services Release 6.1. Private data version 9 provides the following new features:

- Consult Modes
- UCID in Single Step Transfer Call Confirmation event

Consult Modes

Beginning with private data version 9, private data associated with the Held, Service Initiated, and Originated events includes a `consultMode` parameter to indicate whether these events are associated with a conference operation, a transfer operation, or a Consultation Call service request.

When these events are associated with a conference operation initiated at the telephone set (i.e., the user initiates a conference by pressing the Conference button and dialing the added party), the consult mode is set to `ATT_CM_CONFERENCE`.¹³

When these events are associated with a transfer operation initiated at the telephone set (i.e., the user initiates a transfer by pressing the Transfer button and dialing the transfer destination), the consult mode is set to `ATT_CM_TRANSFER`.¹⁴

When an application invokes the Consultation Call service, the consult mode in the Originated event private data is set to `ATT_CM_CONSULTATION`.

UCID in Single Step Transfer Call Confirmation event

Beginning with private data version 9, the confirmation event for the Single Step Transfer Call escape service includes a `ucid` parameter to provide the Universal Call ID (UCID) of the merged call.

¹³ For DCP and H.323 stations, this capability requires Avaya Communication Manager Release 6.0.1 with Service Pack 1, or later. For SIP endpoints, this capability requires Avaya Communication Manager Release 6.2 or later.

¹⁴ For DCP and H.323 stations, this capability requires Avaya Communication Manager Release 6.0.1 with Service Pack 1, or later. For SIP endpoints, this capability requires Avaya Communication Manager Release 6.2 or later.

Private Data Version 9 features, services, and events

The following table maps the Private Data Version 9 features to the services and events that they affect.

Table 27: Private Data Version 9 features, services, and events

Private Data Version 9 feature	Updated services and events
Consult Modes	• Held Event on page 802 • Originated Event on page 827 • Service Initiated Event on page 848
UCID in Single Step Transfer Call Confirmation event	• Single Step Transfer Call (Private Data Version 8 and later) on page 419

Private Data Version 8 features

The TSAPI Service supports private data version 8 beginning with AE Services Release 4.0. Private data version 8 provides the following new features:

- Single Step Transfer Call escape service
- Calling Device in Failed Event

Single Step Transfer Call Escape Service

Normally, performing an unsupervised transfer requires at least two steps:

- **The application invokes the Consultation Call service to place the original call on hold and place a consultation call to the transfer destination.**
- **The application invokes the Transfer Call service to transfer the held call.**

The Single Step Transfer Call escape service allows an application to perform an unsupervised transfer using a single service request.

Calling Device in Failed Event

Beginning with private data version 8, the `ATTFailedEvent_t` includes a `callingDevice` parameter to identify the calling device for the failed call.

Private Data Version 8 features, services, and events

The following table maps the Private Data Version 8 features to the services and events that they affect.

Table 28: Private Data Version 8 features, services, and events

Private Data Version 8 feature	Updated services and events
Single Step Transfer Call Escape Service	• Single Step Transfer Call (Private Data Version 8 and later) on page 419
Calling Device in Failed Event	• Failed Event on page 789

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 Device Name for Attendants](#) on page 1036.
- Enhanced Get API Capabilities function – see [CSTA Get API Capabilities for Private Data](#) Version 7 on page 1037.
- Expanded list of Auxiliary Work Reason codes – see [Increased Aux Reason Codes](#) on page 1037.

Network Call Redirection for Routing

The Adjunct Route support for Network Call Redirection capability allows an application 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 in the route-select message. For the list of TSAPI messages that this feature affects, see [Table 29: Private Data Version 7 features](#) on page 1038.

Redirecting Number Information Element (presented through DeviceHistory)

The “ISDN Redirecting Number for ASAI Events” Communication Manager feature allows 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 [Table 29: Private Data Version 7 features](#) on page 1038.

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 155.

Query Device Name for Attendants

The private Query Device Name service allows an application to query the switch to identify the Integrated Directory name assigned to an extension.

When a name has been assigned to an Attendant station extension, and an application issues a Query Device Name service request, the `deviceType` parameter in the confirmation event 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.

CSTA Get API Capabilities for Private Data Version 7

The `cstaGetAPICaps()` function is enhanced to return the following information.

- Administered Switch Version
- Software Version
- Offer Type (deprecated)
- Server Type (more values to be added in future releases of TSAPI Service)
 - Valid values for Linux systems include: `s8300c`, `s8300d`, `icc`, `premio`, `tn8400`, `laptop`, `CtiSmallServer`, `ibmx306`, `ibmx306m`, `dell1950`, `xen`, `hs20`, `hs20_8832_vm`, `CtiMediumServer`, `isp2100`, `bimx305`, `dl380g3`, `dl385g1`, `dl385g2`, `unknown`, and `CtiLargeServer`
- the maximum number of device history entries (`deviceHistoryCount`)

For the list of TSAPI messages that this feature affects, see [Table 29: Private Data Version 7 features](#) on page 1038.

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 accommodate the new wider value (previously the range was 0-9). For the list of TSAPI messages that this feature affects, see [Table 29: Private Data Version 7 features](#) on page 1038.

Private Data Version 7 features, services, and events

The following table maps the Private Data Version 7 features to the services and events that they affect.

Table 29: Private Data Version 7 features

Private Data Version 7 feature	Updated services and events
Network Call Redirection for Routing	• Conferenced Event on page 644 • Connection Cleared Event on page 668 • Delivered Event on page 678 • Diverted Event on page 725 • Established Event on page 754 • Failed Event on page 789 • Network Reached Event on page 818 • Queued Event on page 838 • Transferred Event on page 854 • Route Select Service (TSAPI Version 2) on page 936
ISDN Redirecting Number Information Element	• Conferenced Event on page 644 • Connection Cleared Event on page 668 • Delivered Event on page 678 • Diverted Event on page 725 • Established Event on page 754 • Failed Event on page 789 • Network Reached Event on page 818 • Queued Event on page 838 • Transferred Event on page 854 • Route Select Service (TSAPI Version 2) on page 936 • Snapshot Call Service on page 549
Query Device Name for Attendants	• Query Device Name Service on page 504
Enhanced GetAPICaps Version	• CSTA Get API Capabilities confirmation structures for Private Data
Increased Aux Reason Codes	• Logged Off Event on page 807 • • • • Query Agent State Service on page 483 • Set Agent State Service on page 439

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 private data confirmation event has been updated to include the fields described in [Table 30](#). See also, [Get API Capabilities Private Data Version 7](#) on page 1040.

Table 30: New ATTV7GetAPICapsConfEvent fields	
New field	Description
<code>char adminSoftwareVersion[256];</code>	Administered switch software version. For example, if the switch version is administered to be 15, then 15 will be passed in the connection accepted message
<code>char softwareVersion[256];</code>	Actual switch software version-- the same software version string that is shown when a customer logs into a SAT for a switch
<code>char offerType[256];</code>	Offer type. Deprecated.
<code>char serverType[256];</code>	Server type. Valid values for Linux systems include: s8300c, s8300d, icc, premio, tn8400, laptop, CtiSmallServer, ibmx306, ibmx306m, dell11950, xen, hs20, hs20_8832_vm, CtiMediumServer, isp2100, bimx305, dl380g3, dl385g1, dl385g2, unknown, and CtiLargeServer
<code>unsigned char deviceHistoryCount</code>	The maximum length for a device history list. For AE Services, this value is 1.

Get API Capabilities Private Data Version 7 Syntax

```

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

typedef struct ATTV7GetAPICapsConfEvent_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   monitorCallsViaDevice;
    unsigned char   deviceHistoryCount;
    char            adminSoftwareVersion[256];
    char            softwareVersion[256];
    char            offerType[256];
    char            serverType[256];
} ATTV7GetAPICapsConfEvent_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 31](#) 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 31: 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

Table 31: Private Data Summary

Private Data Feature	Initial DEFINITY or Communication Manager Release	Initial DEFINITY PBX Driver Release	Initial Private Data Version
Requested Digit Selection	G3V3	R2.1 (private data)	V2
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 <code>cstaGetAPICaps</code> Confirmation Private Data	G3V3	R2.2 (private data)	V4
Support Detailed <code>DeviceIDType_t</code> 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	G3V4	R3.10 (private data)	V5

Table 31: Private Data Summary

Private Data Feature	Initial DEFINITY or Communication Manager Release	Initial DEFINITY PBX Driver Release	Initial Private Data Version
Request			
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
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

Table 31: Private Data Summary

Private Data Feature	Initial DEFINITY or Communication Manager Release	Initial DEFINITY PBX Driver Release	Initial Private Data Version
User-to-User Info (UUI) is increased from 32 to 96 bytes	G3V8	R3.30 (private data)	V6

Table 32: Renaming PDUs and structures – Private Data Version 13

If your code contains these PDUs and structure member names	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	ATTV12_DELIVERED ATTV12DeliveredEvent_t V12deliveredEvent
ATT_ESTABLISHED ATTEstablishedEvent_t establishedEvent	ATTV12_ESTABLISHED ATTV12EstablishedEvent_t V12establishedEvent
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_t routeRequest	ATTV6_ROUTE_REQUEST ATTV6RouteRequestEvent_t v6routeRequest
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 33: 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 private data version 6	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

Summary of private data versions 2 through 6

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
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, the TSAPI Service supports only a limited number of types of `DeviceIDType_t` for the `deviceIDType` parameter of an `ExtendedDeviceID_t`. The types supported are described in CSTA Device Type and Status (Private Data Version 4 and Earlier).

Table 34: CSTA Device Type and Status (Private Data Version 4 and Earlier)			
<code>DeviceIDType_t</code>	<code>ConnectionID_Device_t</code>	<code>DeviceIDStatus_t</code>	Type of Devices
<code>DEVICE_IDENTIFIER</code>	<code>STATIC_ID</code>	<code>ID_PROVIDED</code>	Internal or external endpoints that have a known device identifier
<code>TRUNK_IDENTIFIER</code>	<code>DYNAMIC_ID</code>	<code>ID_PROVIDED</code>	Internal or external endpoints that do not have a known device identifier
<code>EXPLICIT_PUBLIC_UNKNOWN</code>		<code>ID_NOT_KNOWN</code> or <code>ID_NOT_REQUIRED</code>	

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](#)
- [Private Data Version 4 Syntax](#)

Get API Capabilities Private Data Version 5 and 6 Syntax

```

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

typedef struct ATTV6GetAPICapsConfEvent_t
{
    char                switchVersion[16];
    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       reserved1;
    unsigned char       reserved2;
} ATTV6GetAPICapsConfEvent_t;

```

Get API Capabilities Private Data Version 4 Syntax

```

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

typedef struct ATTV4GetAPICapsConfEvent_t
{
    char                switchVersion[16];
    unsigned char       sendDTMFTone;
    unsigned char       enteredDigitsEvent;
    unsigned char       queryDeviceName;
    unsigned char       queryAgentMeas;
    unsigned char       querySplitSkillMeas;
    unsigned char       queryTrunkGroupMeas;
    unsigned char       queryVdnMeas;
    unsigned char       reserved1;
    unsigned char       reserved2;
} ATTV4GetAPICapsConfEvent_t;

```

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(
    ATTPrivateData_t      *privateData,
    ATTWorkMode_t         workMode,
    long                  reasonCode);

/* attSetAgentStateExt() - Private Data V6 and LaterInterface */

RetCode_t attV6SetAgentState(
    ATTPrivateData_t      *privateData,
    ATTWorkMode_t         workMode,
    long                  reasonCode,
    Boolean                enablePending);
```

Private Data Sample Code

This section provides the following examples of Private Data sample code:

- Sample Code 1 – Direct-Agent Make Call Service
- Sample Code 2 – Set Agent State to Log In with Initial Work Mode Auto-In
- Sample Code 3 – Query ACD Split escape service

Sample Code 1

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

/*
 * Make Direct Agent Call - from "1000" to ACD Agent extension "1001"
 *   - ACD agent must be logged into split "2000"
 *   - 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 = "1000";   /* Call originator, an on-PBX
                                         *
extension */
DeviceID_t   called  = "1001";   /* Call destination, an ACD
                                         *
Agent extension */
DeviceID_t   split   = "2000";   /* ACD Agent is logged into
                                         *
this split */
Boolean      priorityCall = FALSE; /* Not a priority call */
RetCode_t    rc;                /* Return code for
service                                         *

requests */
CSTAEvent_t   cstaEvent;         /* CSTA event buffer */
unsigned short eventBufSize;      /* CSTA event buffer size */
unsigned short numEvents;         /* Number of events queued */
ATTPrivateData_t privateData;    /* ATT service request private
                                         *
data buffer */

/* Format private data for the subsequent cstaMakeCall() request */
rc = attDirectAgentCall(&privateData, &split, priorityCall, NULL);

if (rc < 0)
{
    /* Some kind of failure, handle error here. */
}
```

```

/* Invoke cstaMakeCall() with the formatted private data */
rc = cstaMakeCall(acsHandle, invokeID, &calling, &called,
                  (PrivateData_t *)&privateData);

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/* cstaMakeCall() succeeded. Wait for the confirmation event. */

/* Initialize buffer sizes before calling acsGetEventBlock() */
eventBufSize = sizeof(cstaEvent);
privateData.length = ATT_MAX_PRIVATE_DATA;

rc = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
                     &eventBufSize, (PrivateData_t *)&privateData, &numEvents);

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/* Is this the event that we are waiting for? */
if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
    (cstaEvent.eventHeader.eventType == CSTA_MAKE_CALL_CONF))
{
    if (cstaEvent.event.cstaConfirmation.invokeID == 1)
    {
        /* Invoke ID matches, cstaMakeCall() is confirmed. */
    }
    else
    {
        /* Wrong invoke ID, need to wait for another event */
    }
}
else
{
    /* Wrong event, need to wait for another event */
}

```

Sample Code 2

```

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

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

ACSHandle_t acsHandle; /* An opened ACS
Stream Handle */
InvokeID_t invokeID = 1; /* Application-
generated invoke

    * ID */
DeviceID_t device = "1000"; /* Device associated with
ACD

    * agent */
AgentMode_t agentMode = AM_LOG_IN; /* Requested Agent Mode */
AgentID_t agentID = "3000"; /* Agent login identifier */
AgentGroup_t agentGroup = "2000"; /* ACD split to log Agent into */
AgentPassword_t *agentPassword = NULL; /* No password */
RetCode_t rc; /* Return code
for service

    * requests */
CSTAEvent_t cstaEvent; /* CSTA event buffer
*/
unsigned short eventBufSize; /* CSTA event buffer
size */
unsigned short numEvents; /* Number of events
queued */
ATTPrivateData_t privateData; /* ATT service request
private

    * data buffer */
ATTEvent_t attEvent; /* Private data event
structure */

/*
 * Format private data for the subsequent cstaSetAgentState() request
 */
rc = attV6SetAgentState(&privateData, WM_AUTO_IN, 0, TRUE);

if (rc < 0)
{
    /* Some kind of failure, handle error here. */
}

/* Invoke cstaSetAgentState() with the formatted private data */
rc = cstaSetAgentState(acsHandle, invokeID, &device, agentMode,
    &agentID, &agentGroup, agentPassword,
    (PrivateData_t *)&privateData);

```

CSTAGetAPICaps Confirmation interface structures for Private Data Versions 4, 5, and 6

```

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/* cstaSetAgentState() succeeded. Wait for the confirmation event. */

/* Initialize buffer sizes before calling acsGetEventBlock() */
eventBufSize = sizeof(cstaEvent);
privateData.length = ATT_MAX_PRIVATE_DATA;

rc = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
    &eventBufSize, (PrivateData_t *)&privateData, &numEvents);

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/* Is this the event that we are waiting for? */
if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
    (cstaEvent.eventHeader.eventType == CSTA_SET_AGENT_STATE_CONF))
{
    if (cstaEvent.event.cstaConfirmation.invokeID == 1)
    {
        /* Invoke ID matches, cstaSetAgentState() is confirmed. */

        /* See if the confirmation event includes private data. */
        if (privateData.length > 0)
        {
            /*
             * The confirmation event contains private data.
             * Decode it.
             */
            if (attPrivateData(&privateData, &attEvent) !=

                                ACSPOSITIVE_ACK)
            {
                /* Handle decoding error here. */
            }

            if (attEvent.eventType == ATT_SET_AGENT_STATE_CONF)
            {
                /*
                 * See whether the requested change is pending
                 */
                ATTSetAgentStateConfEvent_t
*setAgentStateConf;
                setAgentStateConf =
&privateData.u.setAgentState;
                if (setAgentStateConf->isPending == TRUE)
                {
                    /* The request is pending */
                }
            }
        }
    }
}

```

}

Sample Code 3

```

#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  device = "1000";    /* Device associated with ACD
                                */

agent */
RetCode_t   rc;                 /* Return code for
service
                                */

requests */
CSTAEvent_t  cstaEvent;         /* CSTA event buffer */
unsigned short eventBufSize;    /* CSTA event buffer size */
unsigned short numEvents;       /* Number of events queued
*/
ATTPrivateData_t privateData;    /* ATT service request private
                                */

data buffer */
ATTEvent_t   attEvent;          /* Private data event
structure */
ATTQueryAcdSplitConfEvent_t    /* Query ACD Split confirmation
    *queryAcdSplitConf;         * event pointer */

/*
 * Format private data for the subsequent cstaEscapeService() request
 */
rc = attQueryAcdSplit(&privatedata, &deviceID);

if (rc < 0)
{
    /* Some kind of failure, handle error here. */
}

/* Invoke cstaEscapeService() with the formatted private data */
rc = cstaEscapeService(acsHandle, invokeID,
    (PrivateData_t *)&privateData);

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/*
 * cstaEscapeService() succeeded. Now wait for the confirmation
event.

```


CSTAGetAPICaps Confirmation interface structures for Private Data Versions 4, 5, and 6

```

*
* To retrieve private data accompanying the confirmation event,
* the application must provide a pointer to a private data buffer as
* a parameter to either an acsGetEventBlock() or acsGetEventPoll()
* request. After receiving an event, the application passes the
* address of the private data buffer to attPrivateData() for
decoding.
*/

/* Initialize buffer sizes before calling acsGetEventBlock() */
eventBufSize = sizeof(cstaEvent);
privateData.length = ATT_MAX_PRIVATE_DATA;

rc = acsGetEventBlock(acsHandle, (void *)&cstaEvent,
                     &eventBufSize, (PrivateData_t *)&privateData, &numEvents);

if (rc != ACSPOSITIVE_ACK)
{
    /* Some kind of failure, handle error here. */
}

/* Is this the event that we are waiting for? */
if ((cstaEvent.eventHeader.eventClass == CSTACONFIRMATION) &&
    (cstaEvent.eventHeader.eventType == CSTA_ESCAPE_SVC_CONF))
{
    if (cstaEvent.event.cstaConfirmation.invokeID == 1)
    {
        /* Invoke ID matches, cstaEscapeService() is confirmed. */

        /* See if the confirmation event includes private data. */
        if (privateData.length > 0)
        {
            /*
             * The confirmation event contains private data.
             * Decode it.
             */
            if (attPrivateData(&privateData, &attEvent) !=

                               ACSPOSITIVE_ACK)
            {
                /* Handle decoding error here. */
            }

            if (attEvent.eventType == ATT_QUERY_ACD_SPLIT_CONF)
            {
                queryAcdSplitConf =
                    (ATTQueryAcdSplitConfEvent_t *)
                     &attEvent.u.queryAcdSplit;
                /* Process event field values here */
            }
        }
        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 35](#) provides Communication Manager CSTA System Capacities. These are maximum system capacities. The defined capacities, as well as the server's hardware configuration and the switch configuration, limit the capacity of the TSAPI Service.

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.

Refer to the *Avaya Aura® Communication Manager System Capacities Table* (<http://support.avaya.com/css/P8/documents/100092572>), document number 03-300511, for Communication Manager system capacities.

Table 35: Communication Manager System Capacities	
Feature Name	Comments
Communication Manager servers supported by one AES server	Defines the number of switch connections that may be administered on AE Services. AE Services allows a TSAPI CTI link to be administered for each switch connection
AES Servers per Communication Manager	Defines the total number of AE Services server connections that may be active simultaneously.
Adjunct Control Associations per Call	Defines the maximum number of monitors per call. Note that certain TSAPI services (e.g., Make Predictive Call, Clear Call, Selective Listening Hold, and Selective Listening Retrieve) may initiate a call monitor even though a monitor was not explicitly requested by any application. The TSAPI Service multiplexes cstaMonitorCall() requests for the same call into a single association, so the CM maximum does not come into play when there is a single AE Services server and the TSAPI Service is the only service monitoring the call. However, when multiple AE Services servers are trying to monitor the same call, or when the CVLAN and/or DLG services are also trying to monitor the same call, the CM maximum may be reached.
Active Adjunct Control Associations	Defines the total number of calls that may be monitored simultaneously.
Active Adjunct Route Requests	Defines the total number of call routing sessions that may be active simultaneously.

Table 35: Communication Manager System Capacities	
Feature Name	Comments
Active Notifications per Split Domain	Defines the maximum number of calls-via-device monitors per ACD split. The TSAPI Service multiplexes <code>cstaMonitorCallsViaDevice()</code> requests for the same ACD split into a single association, so the CM maximum does not come into play when there is a single AE Services server and the TSAPI Service is the only service monitoring the ACD split. However, when multiple AE Services servers are trying to monitor the same ACD split, or when the CVLAN and/or DLG services are also trying to monitor the same ACD split, the CM maximum may be reached.
Active Notifications per VDN Domain	Defines the maximum number of calls-via-device monitors per VDN. The TSAPI Service multiplexes <code>cstaMonitorCallsViaDevice()</code> requests for the same VDN into a single association, so the CM maximum does not come into play when there is a single AE Services server and the TSAPI Service is the only service monitoring the VDN. However, when multiple AE Services servers are trying to monitor the same VDN, or when the CVLAN and/or DLG services are also trying to monitor the same VDN, the CM maximum may be reached.
Domain-Control Associations per Call	Defines the maximum number of monitored stations that may be involved in a call.
3rd-party Domain-Control Station Associations	Defines the maximum number of station monitors that may be active simultaneously. Note that many of the TSAPI call control services (e.g., Make Call, Answer Call, Hold Call, etc.) may initiate a station monitor even though a monitor was not explicitly requested by any application. The TSAPI Service may also initiate station monitors that were not explicitly requested by any application in order to obtain accurate call state information.

Table 35: Communication Manager System Capacities	
Feature Name	Comments
Domain-Control Split/Skill Associations	Defines the total number of Splits/Skills that may be monitored simultaneously
Domain-controllers per Station Domain	Defines the maximum number of device monitors per station. The TSAPI Service multiplexes cstaMonitorDevice() requests for the same station into a single association, so the CM maximum does not come into play when there is a single AE Services server and the TSAPI Service is the only service monitoring the station. However, when multiple AE Services servers are trying to monitor the same station, or when the CVLAN and/or DLG services are also trying to monitor the same station, the CM maximum may be reached.
Domain-controllers per Split/skill Domain	Defines the maximum number of device monitors per ACD split/skill. The TSAPI Service multiplexes cstaMonitorDevice() requests for the same ACD split/skill into a single association, so the CM maximum does not come into play when there is a single AE Services server and the TSAPI Service is the only service monitoring the ACD split/skill. However, when multiple AE Services servers are trying to monitor the same ACD split/skill, or when the CVLAN and/or DLG services are also trying to monitor the same ACD split/skill, the CM maximum may be reached.
Event Notification Associations	Defines the total number of ACD split/skills and VDNs that may be monitored simultaneously using cstaMonitorCallsViaDevice().
Max Calls With Send DTMF Active	Defines the maximum number of calls where the Send DTMF Tones service may be used simultaneously.

Table 35: Communication Manager System Capacities	
Feature Name	Comments
Max Simultaneous Calls Being Classified	Defines the total number of calls that can be classified simultaneously. Note that certain TSAPI services, such as the Make Predictive Call service, use call classifiers.
Simultaneous Selective Listening Disconnected Paths	Defines the total number of call participants that can be on Selective Hold simultaneously.
Maximum ASAI Links (Open and Proprietary)	Defines the maximum number of CTI links that may be administered on CM.

Index

*		
* and # characters		
send DTMF tone	344	
A		
AAR/ARS		
make call	284	
Abbreviated dialing		
originated event	751	
Account codes		
originated event	751	
ACD destination		
make call	284	
ACD group		
device type	140	
ACD originator		
make call	284	
ACD split		
monitor calls via device	532	
monitor device	544	
Ack parameters		
alternate call	217	
answer call	220	
change monitor filter	510	
change system status filter	888	
clear call	224	
clear connection	227	
conference call	234	
consultation call	242	
consultation direct-agent call	253	
consultation supervisor-assist call	263	
conventions	15	
deflect call	271	
hold call	276	
make call	282	
make direct-agent call	295	
make predictive call	307	
make supervisor-assist call	318	
monitor call	519	
monitor calls via device	531	
monitor device	542	
monitor stop	556	
monitor stop on call	552	
pickup call	325	
query ACD split	414	
query agent login	419	
query agent state	426	
query call classifier	434	
query device info	438	
query device name	445	
query do not disturb	452	
query endpoint registration info	456	
query forwarding	463	
query message waiting indicator	467	
query station status	472	
query time of day	476	
query trunk group	480	
query UCID	484	
reconnect call	331	
retrieve call	337	
route end service (TSAPI v2)	814	
route register	824	
route register cancel	820	
route select (TSAPI v2)	851	
selective listening hold	349	
selective listening retrieve	355	
send DTMF tone	342	
set advice of charge	382	
set agent state	388	
set billing rate	398	
set do not disturb	402	
set forwarding feature	406	
set MWI feature	410	
single step conference call	361	
single step transfer call	368	
snapshot call	490	
snapshot device	496	
system status request	869	
system status start	877	
system status stop	884	
transfer call	375	
Ack private parameters		
change monitor filter	510	
change system status filter	889	
conference call	234	
consultation call	242	
consultation direct-agent call	254	
consultation supervisor-assist call	263	
conventions	15	
make call	283	
make direct-agent call	296	
make predictive call	307	
make supervisor-assist call	318	
monitor call	519	
monitor calls via device	531	
monitor device	542	
monitor stop on call	553	
query ACD split	414	
query agent login	419	

- query agent state 427
- query call classifier 434
- query device info 439
- query device name 446
- query endpoint registration info 456
- query message waiting indicator.....467
- query station status.....472
- query time of day 476
- query trunk group.....480
- query UCID.....484
- set advice of charge.....382
- set agent state 388
- single step conference call 361
- single step transfer call.....369
- snapshot device.....496
- system status request.....870
- system status start 878
- transfer call.....375
- ACS.....51
 - Unsolicited Events 106
- ACS Data Types 76, 106, 110, 123, 126, 129, 663, 737
 - Common 111
 - Event.....114
- ACS parameter syntax 18
- ACS stream
 - Aborting.....52, 53, 55
 - Access 53
 - Checking establishment of 53
 - Closing 52, 53, 54
 - CSTA services available on.....52
 - Freeing associated resources 54, 55
 - Opening 51, 52, 53
 - Per advertised service 32
 - Receiving events on.....54
 - Releasing.....53
 - Sending requests and responses over 56
 - set advice of charge.....382
- ACS universal failure events 901
- acsAbortStream().....55, 78
- acsCloseStream() 54, 55, 72
- ACSCloseStreamConfEvent.....54, 74
- acsEnumServerNames().....93
- acsErrorString().....103
- acsEventNotify()
 - Windows 88
- acsFlushEventQueue() 91
- acsGetEventBlock().....57, 79
- acsGetEventPoll() 58, 82
- acsGetFile() (Linux) 85
- acsGetServerID().....95
- acsHandle 53, 54, 55, 56, 57, 58
 - Freeing 54
- acsOpenStream() 53, 63, 97, 98
- ACSOpenStreamConfEvent.....53, 56, 57, 70
- acsQueryAuthInfo() 96, 97, 98
- acsReturnCodeString().....104
- acsReturnCodeVerboseString() 105
- acsSetESR() 58
 - Windows 86
- acsSetHeartbeatInterval().....99
- ACSUniversalFailureConfEvent 76
- ACSUniversalFailureEvent.....76, 106
 - Possible values.....107
- Activation
 - set forwarding feature.....407
- Active state
 - reconnect call 339
 - retrieve call 339
- Adjunct messages
 - set MWI feature.....411
- Adjunct-controlled splits
 - monitor calls via device 532
- Administration 52
- Administration without hardware
 - deflect call.....273
 - monitor device.....544
 - pickup call 328
- Advertised services
 - Getting list of available 51
- Advice of charge event report
 - monitor call.....521
- Agent event filters 507
- AgentMode service parameter.....391
- Alternate call
 - ack parameters 217
 - description 216
 - detailed information.....218
 - functional description.....216
 - nak parameters.....217
 - overview 207
 - service parameters 217
 - syntax.....219
- Analog ports
 - monitor device.....544
- Analog sets 793
- Analog station operation
 - alternate call 222
 - answer call 222
 - reconnect call 222
- Analog stations
 - alternate call 277
 - clear connection 229
 - conference call.....235
 - consultation call.....277
 - hold call.....277

- make call 284
- reconnect call 229
- transfer call 377
- ANI screen pop application requirements 794
- Announcement destination
 - make call 285
- Announcements 794, 797
 - selective listening hold 351
 - selective listening retrieve 351
- Answer call
 - ack parameters 220
 - analog station operation 222
 - description 220
 - detailed information 218, 221, 333
 - nak parameters 221
 - overview 207
 - service parameters 220
 - syntax 223
- Answer supervision timeout 795
- API capabilities
 - private data v4 syntax 952
 - private data v5-6 syntax 952
- API Control Services See ACS
- Application Programming Interface Control
 - Services See ACS
- Applications 51
 - designing using original call info 38
 - designing with screen pop information 35
 - remote, passing UUI 40
- AT&T MultiQuest 900 Vari-A-Bill 397
- Attendant auto-manual splitting 796
- Attendant call waiting 796
- Attendant control of trunk group access 796
- Attendant groups 795
 - monitor device 544
- Attendant specific button operation 795
- Attendants 795
 - deflect call 273
 - make call 285
 - monitor device 544
 - pickup call 328
 - selective listening hold 351
 - selective listening retrieve 351
- AUDIX 796
 - send DTMF tone 344
- Authorization codes
 - make call 285
 - originated event 751
- Auto call back
 - deflect call 273
 - pickup call 328
- Auto-available split 797
- Automatic Call Distribution (ACD) 794, 797
- Automatic callback
 - originated event 751
- B**
- Blind transfer
 - established event 692
- Bridged call appearance 797
 - alternate call 278
 - clear connection 229
 - conference call 235
 - consultation call 278
 - deflect call 273
 - hold call 278
 - make call 285
 - originated event 751
 - pickup call 328
 - reconnect call 229, 339
 - retrieve call 339
 - single step conference call 363
 - transfer call 377
- Busy Hour Call Completions (BHCC)
 - set advice of charge 382
- Busy verification of terminals 798
 - alternate call 278
 - consultation call 278
 - hold call 278
- C**
- Call appearance button 795
- Call classification
 - established event 692
 - make call 285
- Call cleared event 599
 - description 567
 - detailed information 569
 - functional description 567
 - monitor device 540
 - private parameter syntax 571
 - private parameters 569
 - redirection on no answer 793
 - service parameters 568
 - syntax 570
- Call clearing state
 - charge advice event 574
- Call control service group
 - supported services 134
 - unsupported services 137
- Call coverage 798
- Call coverage path containing VDNs 799
 - make call 285
- Call delivered
 - to ACD device 609
 - to ACD split 610
 - to station device 608
 - to VDN 610

- Call destination
 - make call 285
- Call event filters 505
- Call event reports
 - Monitor stop on call 553
- Call forwarding
 - pickup call 328
- Call forwarding all calls 799
 - make call 285
 - set forwarding feature 407
- Call identifier
 - syntax 148
- Call monitoring event sequences
 - single step conference call 363
- Call objects 148
- Call park 799
 - originated event 751
- Call pickup 800
- Call prompting 802
 - for screen pop 35
- Call state 149
 - send DTMF tone 344
 - single step conference call 363
- Call states 497
- Call vectoring 800
 - interactions with feedback 800
 - selective listening hold 351
 - selective listening retrieve 351
- Call waiting 802
 - deflect call 273
 - pickup call 328
- Called number
 - for screen pop 35
- Calling number
 - for screen pop 35
- Calls
 - phantom 139
- Calls In queue, number 803
- Cancel button 796
- Cancel monitor 504
- Capacities
 - system 962
- Change monitor filter
 - ack parameters 510
 - ack private parameters 510
 - description 509
 - detailed information 511
 - functional description 509
 - nak parameters 510
 - overview 503
 - private data v11 and later syntax 514
 - private data v2-4 syntax 516
 - private data v5-10 syntax 515
 - private parameters 510
 - service parameters 509
 - syntax 512
- Change system status filter
 - ack parameters 888
 - ack private parameters 889
 - description 886
 - detailed information 890
 - functional description 886
 - nak parameters 889
 - overview 868
 - private data v2-3 syntax 894
 - private data v4 syntax 893
 - private data v5 and later syntax 892
 - private parameters 887
 - service parameters 886
 - syntax 891
- Charge advice event
 - description 572
 - detailed information 573
 - functional description 572
 - private parameter syntax 576
 - private parameters 572
 - service parameters 572
 - syntax 575
- Charge advice events 541
- Class of Restrictions (COR)
 - make call 285
- Class of Service (COS)
 - make call 285
- Clear call
 - ack parameters 224
 - description 224
 - detailed information 224
 - functional description 224
 - nak parameters 224
 - overview 208
 - service parameters 224
 - syntax 225
- Clear connection
 - ack parameters 227
 - description 226
 - detailed information 229, 333
 - nak parameters 228
 - overview 208
 - private data v6 and later syntax 231
 - private parameters 227
 - service parameters 226
 - syntax 230
 - userInfo parameter 227
- Communication Manager
 - event minimization feature 566
- Communication Manager local call state

mapped to CSTA local call state.....	497	syntax.....	153
Conference	802	Connection state definitions	151, 152
conference call		Consult.....	802
ack private parameters.....	234	Consult mode.....	171, 935
Conference call		held event	725
ack parameters	234	originated event	749
description	233	service initiated event	767
detailed information.....	235	Consultation call	
nak parameters.....	234	ack parameters	242
overview	209	ack private parameters	242
private data v5 and later syntax	238	consult options	168, 241, 932
selective listening hold	351	description	239
selective listening retrieve.....	351	detailed information.....	243
service parameters	233	functional description.....	239
syntax	236	nak parameters.....	242
Conferenced event		overview	209
description.....	577	private data v10 and later syntax	245
detailed information.....	584	private data v2-5 syntax	249
functional description.....	577	private data v6-9 syntax	247
private data v2-3 syntax	597	private parameters	240
private data v4 syntax	595	service parameters	240
private data v5 syntax	592	syntax.....	244
private data v6 syntax	589	Consultation direct-agent call	
private data v7 and later syntax	586	ack parameters	253
private parameters	580	ack private parameters	254
service parameters	578	description	251
syntax	585	detailed information.....	255
trunkList parameter.....	582	functional description.....	251
userInfo parameter	581	nak parameters.....	254
Conferencing call		overview	210
with screen pop information	35	private data v2-5 syntax	259
Conferencing calls		private parameters	253
CSTA services used	36, 38	service parameters	252
Confirmation event		syntax.....	256
format.....	15	Consultation supervisor-assist call	261
Connection cleared event		ack parameters	263
description.....	599	ack private parameters	263
detailed information.....	603	description	261
functional description.....	599	detailed information.....	265
private data syntax	605	nak parameters.....	263
private data v2-5 syntax	607	overview	210
private data v6 syntax	606	private data v2-5 syntax	269
private data v7 and later syntax	605	private data v6 and later syntax	267
private parameters	602	private parameters	262
service parameters	600	service parameters	262
syntax	604	syntax.....	266
userInfo parameter	602	userInfo parameter.....	262
Connection identifier		Consultation transfer	
syntax	150	established event.....	693
Connection identifier conflict	149	Conventions	
Connection object	149	ack parameters	15
Connection state	150	ack private parameters	15
send DTMF tone	344	confirmation event	15

- format.....15
- function 15
- functional description.....16
- nak parameters.....15
- private data 15
- private parameters 15
- service parameters 15
- Converse agent
 - selective listening hold 351
 - selective listening retrieve.....351
- Cover all
 - pickup call 328
- CSTA
 - Confirmation Events 115
 - Control Services.....51, 52, 115
 - Event Data Types 76, 106
 - Services.....53
 - Services available on ACS stream 52
- CSTA local call state
 - mapped to Communication Manager local call state.....497
- CSTA objects
 - call 148
- CSTA services
 - supported 134
 - unsupported 137
- cstaDeflectCall
 - pickup call 328
- cstaErrorString() 128
- CSTAEventCause, values.....559
- cstaGetAPICaps() 52, 116
- CSTAGetAPICapsConfEvent 118
- cstaGetDeviceList() 52, 121
- CSTAGetDeviceListConfEvent 123
- cstaMakePredictiveCall
 - originated event 751
- cstaQueryCallMonitor() 52, 125
- CSTAQueryCallMonitorConfEvent 126
- CTI link failure.....803
- CTI links
 - multiple, considerations for 49
- D**
- Data calls 803
 - make call 286
- Data Types
 - ACS.....110
- DCS
 - make call 286
 - set do not disturb feature 404
 - set forwarding feature.....407
- DCS network, event reporting 803
- Deactivation
 - set forwarding feature.....407
- Deflect call
 - ack parameters 271
 - description 271
 - detailed information.....273
 - functional description.....271
 - nak parameters.....272
 - overview 211
 - service parameters 271
 - syntax.....275
- Deflect from queue
 - deflect call.....273
 - pickup call 328
- Delivered event
 - call coverage path to ACD device 799
 - call scenarios.....620, 623
 - deflect call.....273
 - description 608
 - detailed information.....620
 - functional description.....608
 - last redirection device 620
 - pickup call 328
 - private data syntax 632
 - private data v2-3 syntax 649
 - private data v4 syntax.....646
 - private data v5 syntax.....642
 - private data v6 syntax.....638
 - private data v7 and later syntax 632
 - private parameters 613
 - redirection 793
 - redirection on no answer 793
 - service parameters 610
 - syntax.....631
 - userInfo parameter.....615
- Delivered events
 - consecutive.....609
- Designing applications
 - with screen pop information 35
- Bridged state
 - with bridged state.....498
- Device
 - Query for controllable devices 52
- Device class.....140
- Device ID type
 - private data v2-4.....951
 - private data version 5 and later.....142
- Device identifier.....140
- Device identifiers
 - dynamic 141
 - static 140
- Device monitoring event sequences
 - single step conference call.....363
- Device type
 - ACD group.....140

definitions.....	139
trunk	140
trunk group.....	140
Device types	
station.....	139
Dialing, abbreviated	751
Digits collected	
for screen pop	35
Direct agent calls	
redirection on no answer	793
Direction	
format.....	15
Display	
make call.....	286
make direct-agent call	297
Diverted event	793
ASAI link version 7	653, 658
ASAI link versions 1-6	653
call coverage path (VDNs)	799
calling device	162, 930
deflect call	273
description.....	652
detailed information.....	657
functional description.....	652
new destination.....	657
pickup call	328
private data syntax	660
private data v12 and later syntax	660
private data v7-11 syntax	661
private data version 12.....	162
redirection on no answer	793
service parameters	655
syntax	659
Do Not Disturb	
set do not disturb feature	404
Do not disturb event	7
description.....	662
functional description.....	662
service parameters	662
syntax	663
Drop button	
single step conference call	363
Drop button operation	803
clear connection	229
reconnect call	229
DTMF receiver	
selective listening hold	351
selective listening retrieve.....	351
send DTMF tone	344
DTMF sender	
send DTMF tone	344
DTMF tones, unsupported.....	344
Dynamic device identifier	141

E

Enable pending private parameter	391
EnablePending private parameter	388
En-bloc sets	
service initiated event	804
Endpoint registered event	165
description	664
functional description.....	664
private data v11 and later syntax	668
private parameters	664
service parameters	664
syntax.....	667
Endpoint unregistered event.....	165
description	671
functional description.....	671
private data v11 and later syntax	676
private parameters	671
service parameters	671
syntax.....	675
unregistered reason.....	673
Entered digits event	
description	678
detailed information.....	678
functional description.....	678
private data syntax	680
private parameters	678
service parameters	678
syntax.....	679
Error codes	
TSLIB	908
TSLIB	908
Escape service group	
supported services.....	136
unsupported services	137
Established event	
description	681
detailed information.....	692
functional description.....	681
multiple.....	682
private data syntax	695
private data v2-3 syntax	708
private data v4 syntax.....	705
private data v5 syntax.....	702
private data v6 syntax.....	699
private data v7 and later syntax	695
private parameters	686
service parameters	682
syntax.....	694
userInfo parameter.....	687
Event	
Service Routine (ESR).....	51, 58
Initializing.....	51
Event filters.....	504

agent.....	507	ASAI link version 7	712
call	505	ASAI link versions 1-6.....	712
feature	507	description	711
maintenance.....	507	detailed information.....	717
Event minimization feature		device identifiers	166
on Communication Manager	566	functional description.....	711
Event report service group		private data v7 syntax.....	720
supported services	136	private data v8 and later syntax	719
unsupported services	137	private data version 11	166
Event reports		private parameters	715
detailed information.....	793	service parameters	712
Events	57	syntax.....	718
advice of charge.....	541	Feature access monitoring	
Blocking for	51, 57	monitor device.....	544
call cleared.....	567	Feature event filters	507
charge advice.....	572	Feature summary	
Chronological order	57	for private data	944
conferenced.....	577	Feedback interactions	
connection cleared	599	with call vectoring.....	800
delivered	608	Filters	
diverted	652	agent events	507
do not disturb	662	call events	505
endpoint registered	664	feature events.....	507
endpoint unregistered.....	671	maintenance events	507
entered digits	678	private.....	507
established	681	Forced entry of account codes	
failed.....	711	make call	286
forwarding	721	Formats	
From all streams	57, 58	ack parameters	15
held.....	724	ack private parameters.....	15
logged off.....	729	confirmation event	15
logged on	732	direction.....	15
message waiting	735	function.....	15
monitor ended.....	550	functional description.....	16
network reached	738	nak parameters.....	15
originated	747	private data.....	15
Polling for	51, 57, 58	private parameters	15
Preventing queue overflow	58	service parameters	15
queued.....	756	Forwarded calls	
retrieved	763	deflect call.....	274
route end	809	pickup call	328
route register abort	818	Forwarding event.....	7
route request (TSAPI v1).....	843	description	721
route request (TSAPI v2).....	826	functional description.....	721
route used (TSAPI v1)	866	service parameters	722
route used (TSAPI v2)	862	syntax.....	723
service initiated	766	Functional description	
system status.....	895	conventions	16
system status, overview	868	G	
transferred.....	772	Get API capabilities	
Expert Agent Selection (EAS)	804	private data v11 and later syntax	178
F		private data v4 syntax.....	953
Failed event		private data v5-6 syntax	953

- private data v8-10 syntax 934
- Get API Capabilities
 - private data v7 syntax 942
 - private data v8-10 syntax 934
- H**
- Held event
 - description 724
 - detailed information.....726
 - functional description.....724
 - generating 804
 - private data v9 and later syntax 728
 - private parameters 725
 - service parameters 724
 - switch hook operation.....793
 - syntax 727
- Held state
 - alternate call.....278
 - consultation call.....278
 - hold call 278
- Hold button 795
- Hold call
 - ack parameters 276
 - description.....276
 - detailed information... 218, 243, 255, 265, 277
 - functional description.....276
 - nak parameters.....277
 - overview 211
 - selective listening hold 351
 - selective listening retrieve.....351
 - service parameters 276
 - syntax 279
- Hold state
 - reconnect call 339
 - retrieve call.....339
- Holding calls, generating held event report ...804
- Hot line
 - make call.....286
- I**
- Integrated Services Digital Network (ISDN)....804
- Interactions
 - between feedback and call vectoring.....800
- Interflow 797
- InvokeID*
 - Application generated 56
 - Correlating responses.....56
 - In confirmation event 56
 - In service request* 56
 - Library generated 56
 - Type 56
- ISDN BRI station, single step conference call .359
- L**
- Last added party
 - single step conference call 363
- Last number dialed
 - make call 286
- Last redirection device
 - delivered event 620
 - established event..... 692
 - queued event..... 760
- Links
 - multiple, considerations for..... 49
- Local call states 497
- LocalConnectionInfo parameter
 - monitor services 508
- LocalConnectionState, values 559
- Logged off event
 - description 729
 - detailed information..... 730
 - functional description..... 729
 - private data syntax 731
 - private parameters 729
 - service parameters 729
 - syntax..... 730
- Logged on event
 - description 732
 - detailed information..... 733
 - functional description..... 732
 - private data syntax 734
 - private parameters 732
 - service parameters 732
 - syntax..... 733
- Logical
 - Link..... 52
- Logical agents 804
 - make call 286
 - make direct agent call..... 297
 - monitor device..... 544
 - query endpoint registration info 458
 - set do not disturb feature..... 404
 - set forwarding feature..... 407
- Lookahead interflow 802
- Lookahead interflow info
 - for screen pop..... 35
- Loop back
 - deflect call..... 273
 - pickup call 328
- M**
- Maintenance event filters 507
- Maintenance service group
 - supported services..... 137
 - unsupported services 137
- Make call
 - ack parameters 282
 - ack private parameters..... 283
 - description 280
 - detailed information.....243, 284, 297, 319

functional description.....	280	Maximum requests from multiple AE Services	
nak parameters.....	283	Servers	
overview	211	monitor call.....	521
private data v2-5 syntax	291	Message waiting event	4
private data v6 and later syntax	289	description	735
private parameters	282	detailed information.....	736
service parameters	281	functional description.....	735
syntax	288	service parameters	735
userInfo parameter	282	syntax.....	737
Make direct-agent call		Monitor call	
ack parameters	295	ack parameters	519
description.....	293	ack private parameters.....	519
detailed information.....	255, 297	description	517
functional description.....	293	detailed information.....	521
nak parameters.....	296	functional description.....	517
overview	212	nak parameters.....	520
private data v2-5 syntax	302	overview	503
private data v6 and later syntax	300	private data v11 and later syntax	524
private parameters	295, 296	private data v2-4 syntax	527
service parameters	294	private data v5-10 syntax	525
syntax	299	private parameters	518
Make predictive call		service parameters	518
ack parameters	307	syntax.....	522
ack private parameters.....	307	Monitor calls via device	
description.....	304	ack parameters	531
detailed information.....	310	ack private parameters.....	531
functional description.....	304	description	529
nak parameters.....	308	detailed information.....	532
overview	212	functional description.....	529
private data v2-5 syntax	314	nak parameters.....	532
private data v6 and later syntax	312	overview	503
private parameters	305	private data v11 and later syntax	536
service parameters	305	private data v2-4 syntax	539
syntax	311	private data v5 syntax.....	538
userInfo parameter	306	private data v7-10 syntax	537
Make supervisor-assist call		private parameters	530
ack parameters	318	service parameters	530
ack private parameters.....	318	syntax.....	534
description.....	316	Monitor device	
detailed information.....	265, 319	ack parameters	542
functional description.....	316	ack private parameters.....	542
nak parameters.....	318	description	540
overview	213	detailed information.....	544
private data v2-5 syntax	323	functional description.....	540
private data v6 and later syntax	321	nak parameters.....	543
private parameters	317	overview	504
service parameters	317	private data v11 and later syntax	547
syntax	320	private data v2-4 syntax	549
userInfo parameter	317	private data v5-10 syntax	548
Manual transfer		private parameters	542
established event	693	service parameters	541
Maximum number of monitors		syntax.....	545
monitor calls via device	532	Monitor ended event	

description.....	550	Multiple telephony servers.....	48
detailed information.....	550	Music on hold	
functional description.....	550	alternate call.....	278
overview.....	504	consultation call.....	278
service parameters.....	550	hold call.....	278
syntax.....	551	selective listening hold.....	351
Monitor ended event report		selective listening retrieve.....	351
monitor call.....	521	MWI status sync	
Monitor requests, multiple		set MWI feature.....	411
monitor calls via device.....	532	N	
Monitor service group		Nak parameters	
overview.....	503	alternate call.....	217
supported services.....	135	answer call.....	221
Monitor services		change monitor filter.....	510
localConnectionInfo parameter.....	508	change system status filter.....	889
Monitor stop		clear call.....	224
ack parameters.....	556	clear connection.....	228
description.....	556	conference call.....	234
detailed information.....	557	consultation call.....	242
functional description.....	556	consultation direct-agent call.....	254
nak parameters.....	556	consultation supervisor-assist call.....	263
overview.....	504	conventions.....	15
service parameters.....	556	deflect call.....	272
syntax.....	557	hold call.....	277
Monitor stop on call		make call.....	283
ack parameters.....	552	make direct-agent call.....	296
ack private parameters.....	553	make predictive call.....	308
description.....	552	make supervisor-assist call.....	318
detailed information.....	553	monitor call.....	520
functional description.....	552	monitor calls via device.....	532
nak parameters.....	553	monitor device.....	543
overview.....	504	monitor stop.....	556
private data syntax.....	555	monitor stop on call.....	553
private parameters.....	552	pickup call.....	326
syntax.....	554	query ACD split.....	415
Monitor stop on call service		query agent login.....	419
monitor call.....	521	query agent state.....	428
Multifunction station operation		query call classifier.....	435
alternate call.....	221	query device info.....	439
answer call.....	221	query device name.....	448
reconnect call.....	221	query do not disturb.....	452
Multiple application requests		query endpoint registration info.....	458
monitor call.....	521	query forwarding.....	464
Multiple CLAN connections		query message waiting indicator.....	468
system status request.....	871	query station status.....	472
Multiple events		query time of day.....	477
established event.....	682	query trunk group.....	481
Multiple registered endpoints		query UCID.....	485
query endpoint registration info.....	458	reconnect call.....	332
Multiple requests		retrieve call.....	338
monitor calls via device.....	532	route end service (TSAPI v2).....	814
monitor device.....	544	route register.....	824
Multiple split queueing.....	802, 805	route register cancel.....	820

- route select (TSAPI v2).....852
- selective listening hold 350
- selective listening retrieve.....356
- send DTMF tone 343
- set advice of charge.....382
- set agent state.....388
- set billing rate.....399
- set do not disturb 402
- set forwarding feature.....407
- set MWI feature 410
- single step conference call 362
- single step transfer call.....369
- snapshot call.....492
- snapshot device.....497
- system status request.....870
- system status start 878
- system status stop.....884
- transfer call.....376
- Network reached event
 - description.....738
 - detailed information.....742
 - functional description.....738
 - private data syntax 744
 - private data v2-4 syntax 746
 - private data v5-6 syntax 745
 - private data v7 and later syntax 744
 - private parameters 740
 - service parameters 739
 - syntax 743
- New destination
 - diverted event 657
- Night service.....797
 - make call.....286
- O**
- Objects
 - connection.....149
- Off-PBX destination
 - deflect call 273
 - pickup call.....328
- Original call info
 - for screen pop 38
- Originated event
 - description.....747
 - detailed information.....751
 - functional description.....747
 - private data syntax 753
 - private data v2-5 syntax 755
 - private data v6-8 syntax 754
 - private data v9 and later syntax 753
 - private parameters 749
 - service parameters 748
 - syntax 752
 - userInfo parameter 749
- P**
- Park/unpark call
 - selective listening hold 351
 - selective listening retrieve..... 351
- Party, last added
 - single step conference call..... 363
- Personal Central Office Line (PCOL)..... 805
 - make call 286
 - monitor calls via device 532
 - monitor device..... 544
- Phantom calls 139
 - make call 281
 - make direct-agent call 294
 - make predictive call 305
 - make supervisor-assist call 317
- Pickup call
 - ack parameters 325
 - description 325
 - detailed information..... 328
 - functional description..... 325
 - nak parameters..... 326
 - overview 213
 - service parameters 325
 - syntax..... 329
- PRI
 - make call 286
- Primary old call in conferenced event
 - single step conference call..... 363
- Primary Rate Interface (PRI) 805
- Priority calling
 - make call 286
- Priority calls
 - deflect call..... 274
 - pickup call 328
- Private data
 - feature summary 944
 - sample code..... 179
- Private data features
 - initial PBX Driver release..... 944
 - initial private data version 159, 944
 - initial switch release 944
 - list of 944
- Private data function
 - convention 15
 - format..... 15
- Private data version 10
 - features.....6, 168, 932
- Private data version 11
 - endpoint information query 930
 - endpoint registered event 930
 - endpoint unregistered event..... 931
 - failed event device identifiers 931
 - features.....5, 930

Private Data Version 11 features.....	163	query endpoint registration info	455
Private data version 12		query station status	472
calling device in diverted event	930	query trunk group.....	480
features	5, 930	query UCID.....	484
Private Data Version 12 features.....	162	queued event.....	759
Private data version 6		reconnect call	331
function changes	954	route request event (TSAPI v2).....	828
Private data version 7		route select (TSAPI v2).....	849
features	938	route used event (TSAPI v2)	863
Private data version 8		selective listening hold	349
features	7, 173, 937	selective listening retrieve.....	355
Private data version 9		send DTMF tone	342
features	6, 171, 935	service initiated event	767
Private event parameters		set advice of charge	381
query agent login.....	419	set agent state	387
Private filter.....	507	set billing rate	398
Private Filter, set to On.....	505	single step conference call.....	360
Private parameters		single step transfer call.....	368
call cleared event.....	569	snapshot call	491
change monitor filter	510	system status event.....	896
change system status filter.....	887	system status start.....	877
charge advice event.....	572	transferred event.....	774
clear connection	227		
conferenced event.....	580	Q	
connection cleared event	602	Query	
consultation call.....	240	Call/Call Monitoring.....	52
consultation direct-agent call.....	253	Query ACD split	
consultation supervisor-assist call.....	262	ack parameters	414
conventions	15	ack private parameters.....	414
delivered event.....	613	description	414
endpoint registered event.....	664	nak parameters.....	415
endpoint unregistered event.....	671	private parameter syntax	417
entered digits event	678	private parameters	414
established event	686	service parameters	414
failed event.....	715	syntax.....	416
held event.....	725	Query agent login	
logged off event.....	729	ack parameters	419
logged on event.....	732	ack private parameters.....	419
make call.....	282	description	418
make direct-agent call	295	nak parameters.....	419
make predictive call.....	305	private event parameters	419
make supervisor-assist call	317	private parameter syntax	423
monitor call	518	private parameters	419
monitor calls via device	530	service parameters	418
monitor device	542	syntax.....	421
monitor stop on call	552	Query agent state	
network reached event	740	ack parameters	426
originated event	749	ack private parameters.....	427
query ACD split	414	description	425
query agent login.....	419	detailed information.....	429
query agent state	425	functional description.....	425
query call classifier	434	nak parameters.....	428
query device name	445	private data v2-4 syntax	433
		private data v5 syntax.....	432

private data v6 and later syntax	431	service parameters	455
private parameters	425	syntax.....	459
service parameters	425	Query forwarding	
syntax	430	ack parameters	463
Query call classifier		description	463
ack parameters	434	detailed information.....	464
ack private parameters.....	434	functional description.....	463
description	434	nak parameters.....	464
functional description.....	434	service parameters	463
nak parameters.....	435	syntax.....	465
private data syntax	437	Query message waiting indicator	
private parameters	434	ack parameters	467
service parameters	434	ack private parameters.....	467
syntax	436	description	467
Query device info		detailed information.....	468
ack parameters	438	functional description.....	467
ack private parameters.....	439	nak parameters.....	468
description.....	438	private data syntax	470
detailed information.....	440	service parameters	467
functional description.....	438	syntax.....	469
nak parameters.....	439	Query service group	
private data v2-4 syntax	444	supported services.....	135
Private data v5 and later syntax	443	unsupported services	137
service parameters	438	Query station status	
syntax	441	ack parameters	472
Query device name		ack private parameters.....	472
ack parameters	445	description	471
ack private parameters.....	446	enhanced	169, 471, 933
description.....	445	functional description.....	471
detailed information.....	448	nak parameters.....	472
functional description.....	445	private data v10 and later syntax	474
nak parameters.....	448	private data v2-9 syntax	475
private data v4 syntax	451	private parameters	472
private data v5 and later syntax	450	service parameters	472
private parameters	445	syntax.....	473
service parameters	445	Query time of day	
syntax	449	ack parameters	476
Query do not disturb		ack private parameters.....	476
ack parameters	452	description	476
description.....	452	functional description.....	476
functional description.....	452	nak parameters.....	477
nak parameters.....	452	private data syntax	479
service parameters	452	service parameters	476
syntax	454	syntax.....	478
Query endpoint registration info.....	163	Query trunk group	
ack parameters	456	ack parameters	480
ack private parameters.....	456	ack private parameters.....	480
description.....	455	description	480
detailed information.....	458	functional description.....	480
functional description.....	455	nak parameters.....	481
nak parameters.....	458	private data syntax	483
private data v11 and later syntax	460	private parameters	480
private parameters	455	service parameters	480

syntax	482	service parameters	337
Query UCID		syntax.....	340
ack parameters	484	Retrieved event	
ack private parameters.....	484	description	763
description.....	484	detailed information.....	764
functional description.....	484	functional description.....	763
nak parameters.....	485	service parameters	763
private data syntax	487	switch hook operation	794
private parameters	484	syntax.....	765
service parameters	484	Ringback queueing.....	806
syntax	486	Route end event	
Queued event		description	809
description.....	756	detailed information.....	811
detailed information.....	756, 760	functional description.....	809
functional description.....	756	service parameters	809
private data v7 and later syntax	762	syntax.....	812
private parameters	759	Route end service (TSAPI v1)	
redirection on no answer	793	description	816
service parameters	757	detailed information.....	816
syntax	761	functional description.....	816
Queued event reports, multiple	756	syntax.....	817
R		Route end service (TSAPI v2)	
Reason code private parameter	391	ack parameters	814
Reconnect call		description	813
ack parameters	331	detailed information.....	814
description.....	330	functional description.....	813
detailed information.....	333	nak parameters.....	814
functional description.....	330	service parameters	813
nak parameters.....	332	syntax.....	815
overview	213	Route register	
private data v2-5 syntax	336	ack parameters	824
private data v6 and later syntax	335	description	823
private parameters	331	detailed information.....	824
service parameters	331	functional description.....	823
syntax	333	nak parameters.....	824
userInfo parameter	331	service parameters	824
Recording device, dropping		syntax.....	825
single step conference call	363	Route register abort event	
Release button	796	description	818
Remote agent trunk		detailed information.....	818
single step conference call	363	functional description.....	818
Remote applications, designing for	40	service parameters	818
Requests, multiple		syntax.....	819
monitor device	544	Route register cancel	
Retrieve call		ack parameters	820
ack parameters	337	description	820
description.....	337	detailed information.....	821
detailed information.....	333, 339	functional description.....	820
functional description.....	337	nak parameters.....	820
nak parameters.....	338	service parameters	820
overview	214	syntax.....	822
selective listening hold	351	Route request (TSAPI v2)	
selective listening retrieve.....	351	description	826

Route request event (TSAPI v1)		
description	843	
detailed information	844	
functional description	843	
service parameters	844	
syntax	845	
Route request event (TSAPI v2)		
detailed information	831	
functional description	826	
private data v2-4 syntax	841	
private data v5 syntax	839	
private data v6 syntax	837	
private data v7 and later syntax	834	
private parameters	828	
service parameters	827	
syntax	832	
Route select (TSAPI v1)		
description	860	
detailed information	860	
functional description	860	
syntax	861	
Route select (TSAPI v2)		
ack parameters	851	
description	847	
detailed information	853	
functional description	847	
nak parameters	852	
private data v2-5 syntax	858	
private data v6 syntax	856	
private data v7 and later syntax	854	
private parameters	849	
service parameters	848	
syntax	853	
Route used event (TSAPI v1)		
description	866	
detailed information	867	
functional description	866	
service parameters	866	
syntax	867	
Route used event (TSAPI v2)		
description	862	
detailed information	863	
functional description	862	
private data syntax	865	
private parameters	863	
service parameters	862	
syntax	864	
Routing service group		
supported services	136	
unsupported services	137	
Routing service group, overview	808	
S		
Sample code	955	
Screen pop info		
using original call info	38	
Screen pop information		
called number	35	
calling number	35	
conferencing call	35	
digits collected by call prompting	35	
lookahead interflow information	35	
transferring call	35	
user-to-user information (UII)	35	
Security		
single step conference call	364	
Selective listening hold		
ack parameters	349	
description	348	
detailed information	351, 356	
functional description	348	
nak parameters	350	
private data v5 and later syntax	353	
private parameters	349	
service parameters	348	
syntax	352	
Selective listening retrieve		
ack parameters	355	
description	354	
detailed information	356	
functional description	354	
nak parameters	356	
private data v5 and later syntax	358	
private parameters	355	
service parameters	354	
syntax	357	
Send all calls		
pickup call	328	
Send All Calls (SAC)	806	
make call	286	
set do not disturb feature	404	
Send DTMF tone		
ack parameters	342	
description	341	
detailed information	344	
nak parameters	343	
private data v4 syntax	347	
private data v5 and later syntax	346	
private parameters	342	
service parameters	341	
syntax	345	
Send DTMF Tone		
functional description	341	
Send DTMF tone requests, multiple	344	
Service description		
route request event (TSAPI v1)	844	
Service groups		

call control	134	logged on event	732
escape.....	136	make call	281
event report.....	136	make direct-agent call	294
maintenance.....	137	make predictive call	305
monitor	135	make supervisor-assist call	317
query.....	135	message waiting event	735
routing	136	monitor call.....	518
supported	134	monitor calls via device	530
set feature	134	monitor device.....	541
snapshot	135	monitor ended event	550
system status.....	137	monitor stop	556
Service initiated event		network reached event	739
description.....	766	originated event	748
detailed information.....	768	pickup call	325
functional description.....	766	query ACD split	414
not sent with en-bloc sets	804	query agent login.....	418
private data syntax	770	query agent state.....	425
private data v5-8 syntax	771	query call classifier	434
private data v9 and later syntax	770	query device info	438
private parameters	767	query device name	445
service parameters	766	query do not disturb.....	452
switch hook operation.....	793	query endpoint registration info	455
syntax	769	query forwarding.....	463
Service observing.....	797	query message waiting indicator.....	467
Service parameters		query station status	472
alternate call.....	217	query time of day.....	476
answer call.....	220	query trunk group.....	480
call cleared event.....	568	query UCID.....	484
change monitor filter	509	queued event.....	757
change system status filter.....	886	reconnect call	331
charge advice event.....	572	retrieve call	337
clear call.....	224	retrieved event	763
clear connection	226	route end event	809
conference call	233	route end service (TSAPI v2).....	813
conferenced event.....	578	route register	824
connection cleared event	600	route register abort event	818
consultation call.....	240	route register cancel.....	820
consultation direct-agent call.....	252	route request event (TSAPI v2).....	827
consultation supervisor-assist call.....	262	route select (TSAPI v2).....	848
deflect call	271	route used event (TSAPI v1)	866
delivered event.....	610	route used event (TSAPI v2)	862
diverted event	655	selective listening hold	348
do not disturb event.....	662	selective listening retrieve.....	354
endpoint registered event.....	664	send DTMF Tone	341
endpoint unregistered event.....	671	service initiated event	766
entered digits event	678	set advice of charge	381
established event	682	set agent state	386
failed event.....	712	set billing rate	397
format.....	15	set do not disturb	402
forwarding event	722	set forwarding feature.....	406
held event.....	724	set MWI feature.....	410
hold call	276	single step conference call.....	359
logged off event.....	729	single step transfer call.....	368

snapshot call.....	489	query device info	438
snapshot device.....	496	query device name	445
system status event.....	895	query do not disturb.....	452
system status request.....	869	query endpoint registration info	455
system status start	876	query forwarding.....	463
system status stop.....	884	query message waiting indicator.....	467
transfer call.....	374	query station status.....	471
transferred event.....	773	query time of day.....	476
Service-observing	806	query trunk group.....	480
Services		query UCID.....	484
alternate call.....	216	reconnect call	330
alternate call, overview	207	reconnect call, overview	213
answer call.....	220	retrieve call	337
answer call, overview	207	retrieve call overview	214
change monitor filter.....	509	route end (TSAPI v2)	813
change system status filter.....	886	route end service (TSAPI v1).....	816
change system status filter, overview	868	route register	823
clear call.....	224	route register cancel.....	820
clear call, overview	208	route select (TSAPI v1).....	860
clear connection	226	route select (TSAPI v2).....	847
clear connection, overview	208	selective listening hold	348
conference call	233	selective listening retrieve.....	354
conference call, overview	209	send DTMF tone	341
consultation call.....	239	set advice of charge	381, 541
consultation call, overview	209	set agent state	385
consultation direct-agent call.....	251	set billing rate	397
consultation direct-agent call, overview	210	set do not disturb	402
consultation supervisor-assist call.....	261	set forwarding feature.....	406
consultation supervisor-assist call, overview	210	set MWI feature.....	410
deflect call	271	single step conference call.....	359
deflect call, overview	211	single step conference call, overview	214
hold call	276	single step transfer call.....	368
hold call, overview.....	211	single step transfer call, overview	215
make call.....	280	snapshot call.....	489
make call, overview	211	snapshot device	496
make direct-agent call	293	supported	134
make direct-agent call, overview	212	system status request.....	869
make predictive call.....	304	system status request, overview	868
make predictive call, overview	212	system status start.....	876
make supervisor-assist call	316	system status start, overview	868
make supervisor-assist call, overview	213	system status stop.....	884
monitor call	517	system status stop, overview	868
monitor calls via device	529	transfer call.....	374
monitor device	540	transfer call, overview	215
monitor stop.....	556	unsupported	137
monitor stop on call	552	Set advice of charge.....	541
pickup call.....	325	ack parameters	382
pickup call, overview	213	ack private parameters.....	382
query ACD split	414	description	381
query agent login.....	418	detailed information.....	381, 382
query agent state	425	nak parameters.....	382
query call classifier	434	private parameter syntax	384
		private parameters	381

service parameters	381	ack parameters	361
syntax	383	ack private parameters	361
Set agent state		description	359
ack parameters	388	detailed information	363
ack private parameters	388	functional description	359
description	385	nak parameters	362
detailed information	391	overview	214
functional description	385	private data v5 and later syntax	366
nak parameters	388	private parameters	360
private data v2-4 syntax	396	service parameters	359
private data v5 syntax	395	syntax	365
private data v6 and later syntax	394	Single step transfer call	173
private parameters	387	ack parameters	368
service parameters	386	ack private parameters	369
syntax	392	confirmation event	172
Set billing rate		description	368
ack parameters	398	functional description	368
description	397	nak parameters	369
detailed information	399	private data v8 syntax	373
functional description	397	private data v9 and later syntax	372
nak parameters	399	private parameters	368
private parameter syntax	401	service parameters	368
private parameters	398	syntax	371
service parameters	397	Single Step Transfer Call	
syntax	400	overview	215
Set do not disturb		Single-digit dialing	
ack parameters	402	make call	287
functional description	402	SIP endpoint address	
nak parameters	402	query endpoint registration info	458
service parameters	402	SIP endpoint MAC address	
Set do not disturb feature		query endpoint registration info	458
description	402	SIP station extensions	
detailed information	404	query endpoint registration info	458
syntax	405	SIP Stations	
Set feature service group		make direct-agent call	297
supported services	134	Skill hunt groups	
Set forwarding		make call	287
functional description	406	monitor calls via device	533
Set forwarding feature		monitor device	544
ack parameters	406	Snapshot call	
description	406	ack parameters	490
nak parameters	407	description	489
service parameters	406	functional description	489
syntax	408	nak parameters	492
Set MWI feature		private parameters	491
ack parameters	410	service parameters	489
description	410	Snapshot call service	
detailed information	411	CSTA connection states	489
functional description	410	syntax	493
nak parameters	410	Snapshot device	
service parameters	410	ack parameters	496
syntax	412	ack private parameters	496
Single step conference call		description	496

detailed information.....	497	private data v2-3 syntax	900
functional description.....	496	private data v4 syntax.....	899
nak parameters.....	497	private data v5 and later syntax	898
private data v2-4 syntax	502	private parameters	896
private data v5 and later syntax	501	service parameters	895
service parameters	496	syntax.....	897
syntax	499	System status events	
Snapshot service group		not supported	868
supported services	135	System status group	
Snapsnot call service		unsupported services	138
private data syntax	495	System status request	
Split button.....	796	ack parameters	869
Start button	795	ack private parameters.....	870
State of added station		description	869
single step conference call	364	detailed information.....	871
Static device identifier	140	functional description.....	869
Station		multiple CLAN connections.....	871
device type	139	nak parameters.....	870
monitor calls via device	533	overview	868
Station Message Detail Recording (SMDR)		private data v2-3 syntax	875
make call.....	287	private data v4 syntax.....	874
Subdomain boundary		private data v5 and later syntax	873
switching.....	738	service parameters	869
Switch administration		syntax.....	872
selective listening hold	351	System status service group	
selective listening retrieve.....	351	supported services.....	137
Switch hook operation	793	System status start	
Switch operation		ack parameters	877
retrieve call.....	339	ack private parameters.....	878
Switch operation		description	876
after clear call.....	224	detailed information.....	879
alternate call.....	278	functional description.....	876
clear connection	229	nak parameters.....	878
consultation call.....	278	overview	868
hold call	278	private data v2-3 syntax	883
make call.....	287	private data v4 syntax.....	882
reconnect call	229	private data v5 and later syntax	881
Switch operation		private parameters	877
reconnect call	339	service parameters	876
Switch operation		syntax.....	880
monitor stop.....	557	System status stop	
Switch-hook flash field	377	ack parameters	884
Switching subdomain boundary	738	description	884
Synthesized message retrieval		detailed information.....	884
set MWI feature	411	functional description.....	884
System capacity	962	nak parameters.....	884
System starts		overview	868
set MWI feature	411	service parameters	884
System status event		syntax.....	885
description.....	895	T	
detailed information.....	896	Telephony servers	
functional description.....	895	multiple.....	48
overview	868	Temporary bridged appearance	

- clear connection 229
- reconnect call 229
- Temporary bridged appearances 793, 806
- Terminating Extension Group (TEG) 806
 - make call 287
 - monitor calls via device 533
 - monitor device 544
- Tone cadence and level
 - send DTMF tone 344
- Transfer 807
 - established event 693
- Transfer call
 - ack parameters 375
 - ack private parameters 375
 - description 374
 - detailed information 377
 - functional description 374
 - nak parameters 376
 - overview 215
 - private data v5 and later syntax 379
 - selective listening hold 351
 - selective listening retrieve 351
 - service parameters 374
 - syntax 378
- Transferred event
 - description 772
 - detailed information 778
 - functional description 772
 - private data v2-3 syntax 791
 - private data v4 syntax 789
 - private data v5 syntax 786
 - private data v6 syntax 783
 - private data v7 and later syntax 780
 - private parameters 774
 - service parameters 773
 - syntax 779
 - trunkList parameter 776
 - userInfo parameter 775
- Transferring call
 - with screen pop information 35
- Transferring calls
 - CSTA services used 36, 38
- Troubleshooting
 - ACS universal failure events 901
- Trunk
 - device type 140
- Trunk group
 - device type 140
- Trunk group access 796
- Trunk group administration
 - charge advice event 574
- Trunk to trunk transfer
 - transfer call 377
- TrunkList parameter
 - conferenced event 582
 - transferred event 776
- Trunk-to-trunk transfer 807
- TSLIB
 - error codes 908
- U**
- Unsolicited Events
 - ACS 106
- UserInfo parameter
 - maximum size 227, 262, 282, 306, 317, 331, 581, 602, 615, 687, 775
 - not supported by switch 749
- User-to-user info
 - passing info to remote applications 40
- User-to-user information (UUI)
 - for screen pop 35
- V**
- VDN
 - make call 284, 287
 - monitor device 544
- VDN destination
 - make call 287
- Vector-controlled split
 - monitor calls via device 533
 - monitor device 544
- Voice (synthesized) message retrieval
 - set MWI feature 411
- W**
- Work mode private parameter 391